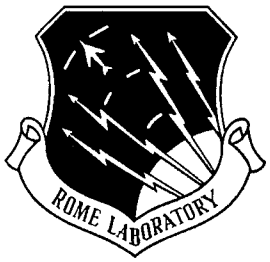
RL-TR-96-22
Final Technical Report
February 1996



FAULT TOLERANCE IN ADM

Key Software, Inc.

Douglas G. Weber

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR-96-22 has been reviewed and is approved for publication.

APPROVED:

DOUGLAS A. WHITE
Project Engineer

FOR THE COMMANDER:

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory/ (C3CA), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1996	Final Sep 93 - Sep 95

4. TITLE AND SUBTITLE

FAULT TOLERANCE IN ADM

5. FUNDING NUMBERS

C - F30602-93-C-0237
PE - 62702F
PR - 5581
TA - 27
WU - 72

6. AUTHOR(S)

Douglas G. Weber

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Key Software, Inc.
840 Hanshaw Rd
Suite 8
Ithaca NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3CA
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-96-22

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Douglas A. White/C3CA/(315) 330-3564

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes research and development undertaken to provide specialized support for developing fault tolerant systems within the Advanced Development Model (ADM) of Rome Laboratory's Knowledge-Based Software Assistant (KBSA). The KBSA provides capabilities for developing software that improve both resultant quality and productivity. In the KBSA, knowledge of techniques used in creating designs and algorithms are encoded in the "knowledge-base" and are applied to provide automated support for system developers. A Concept Demonstration of the KBSA was completed in 1992, and a product-quality version, called the ADM, will be available in 1997. The research and development in fault tolerance is supplementary to the ADM and creates enhancements to the ADM, specifically the specification language. This will enable the refinement of a system specification, written in the ADM specification language, into a transparently distributed, fault tolerant version of the original. The fault tolerance transformations described in this report are based on previous research and depend on distributing redundant components of a system over a network.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software, Fault tolerance, Knowledge-based systems, Program transformation	92
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

Chapter 1

Introduction

1.1 Report Identification

The Fault Tolerance in ADM research project was performed by Key Software, Inc., of Ithaca, New York, for the US Air Force Rome Laboratory over a 2 year period ending September 1995. The contract number for this work was F30602-93-C-0237.

This document is the Final Technical Report for the Fault Tolerance in ADM project. This Final Technical Report describes the results of the entire project. It supersedes an earlier Interim Summary Technical Report that was delivered in 1994.

We assume that readers of this report are familiar with basic terminology and concepts underlying computer systems and computer programming, such as compilation, execution, processors, processes, and networking. We do not assume any prior knowledge of specific programming languages, of fault tolerance, or of the ADM system.

1.2 Project Goals

The goal of this project was to enhance the capability of the Advanced Development Model (ADM) of Rome Laboratory's Knowledge-Based Software Assistant (KBSA). Key Software's enhancement of the KBSA ADM is specialized support for developing fault tolerant systems. In this section we briefly describe KBSA, ADM, and fault tolerance. In the rest of this chapter we outline our accomplishments in enhancing ADM and note some ways in which the work could be continued. The rest of the report describes our ADM enhancements in more detail.

1.2.1 KBSA

The KBSA program aims to find better methods for developing quality software. The following are some key methods under investigation.

- *Process modeling* means describing in detail the actions involved in good software development and, using this description, automating the sequence of actions so that it can be followed consistently and accurately.
- *Program transformation* means modifying simple, abstract programs (or their specifications) into efficient, concrete ones by transforming the program text.

These two methods are complementary. Both methods tend to reduce the number of mistakes commonly made in designing and coding software.

In KBSA the methods of process modeling and program transformation are given intelligent computer support. That support is called "knowledge-based" because it encodes and applies knowledge of techniques that are useful for creating good designs and algorithms. These techniques are discovered by engineers in the course of designing and implementing systems. If a technique can be applied to several similar systems, it can often be reused and can sometimes be automated. The "knowledge" in KBSA is both knowing the techniques and knowing the kind of system to which they can best be applied.

Systems are usually classified in one of two ways:

1. according to the kind of problem they solve, e.g., database management, air traffic control, or scientific computing;
2. according to the critical properties they must have, e.g., fault tolerance, security, or real time performance.

Either of these two ways of classifying can be used to group systems into *domains*, where each domain has specialized engineering techniques. Knowledge-based support, then, is often specific to a domain. Our project will encode knowledge from the fault tolerance domain into KBSA.

A Concept Demo of KBSA was completed in 1992 by a team led by Andersen Consulting of Chicago, Illinois [10]. Andersen Consulting is now leading the development of a production-quality version of the KBSA, called the Advanced Development Model (ADM). ADM will include support not only for the process modeling and

program transformation mentioned previously, but also for machine-aided reasoning about programs and their specifications, and for automatically generating textual documentation. The ADM will be completed in 1996.

Our Fault Tolerance in ADM project is not part of the main ADM development though it proceeded at the same time. Our project was finished roughly a year before ADM. Nevertheless, our results, the enhancements to ADM, were intended to be usable within the ADM framework once that framework is complete. Therefore we have closely monitored the work of the Andersen team and have built on the ADM design decisions as they were made.

1.2.2 Fault Tolerance

A system is *fault tolerant* if it behaves acceptably even when some of its components have malfunctioned. We assume that a system is built from a collection of interacting components. Any component may malfunction, although the probability of a given component's malfunction is limited. A malfunction, or *failure*, is the change in a component's behavior that results from some cause, called a *fault*. A system will tolerate faults if, in spite of failures, its behavior may change but will still remain acceptable. For example, a component failure may not affect the system's functionality, or it may degrade the system's real time performance only minimally, or both. The meaning of fault tolerance depends on what behavior is acceptable, but in every case some important property of the system remains unchanged in the presence of faults and the component failures they cause.

Fault tolerance is a critical requirement that must be addressed in systems of many different kinds, but especially in computer systems that

- are *embedded* as part of an autonomous controller for physical devices, or
- need real time performance that doesn't allow for down-time, or
- are distributed over several processors connected by a network.

Embedded controllers and continuous real time operation clearly need fault tolerance because their applications cannot wait for human operators to repair failed components. The relation with distributivity, however, may be less clear. To see that distributed systems and fault tolerance are closely related, consider the following:

- Adding new components to a system that is distributed over network must inevitably add new ways for that system to fail. A system's reliability will

therefore decrease for larger networks as long as failure of individual components can cause failure of the entire system. Countering this trend, fault tolerance mechanisms build redundancy into the system, making it independent of one or more failures in network hardware. Fault tolerance, therefore, is essential to the design of distributed systems.

- Conversely, designing fault tolerance into a system often means that the system *should* be distributed across a network: distribution makes a failure of one system component less likely to be correlated with the failure of other components elsewhere in the network. Thus when one component fails it is likely that other redundant components do not also fail at the same time; it is this independence of failures on which fault tolerance depends. For example, sharing power supplies and electrical groundings can lead to correlated hardware failures, and thus can lead to a single point of failure for the system they support. Physically distributing hardware is the simplest way to avoid this correlation.

The fault tolerance techniques described in this report all depend on distributing redundant components of a system over a network.

Fault tolerance fits well into the KBSA paradigm because often a fault tolerant system is a modification of a fault-intolerant system that performs the same functions. The fault tolerant system has additional, transparent (i.e., hidden), mechanisms for coping with component failures. Modifying a system to introduce these fault tolerance mechanisms can be thought of as an instance of the KBSA program transformation method. The fault tolerance transformations we describe for ADM will refine a system specification, written in the ADM specification language, into a transparently distributed, fault tolerant version of the original.

The fault tolerance transformations developed on this project are based on previous research. Fault tolerance mechanisms have been studied and used for decades. Most research and development of fault tolerant distributed systems has used an approach called *N-modular redundancy* in which system components are replicated, distributed, and the behavior of the replicas is coordinated to tolerate failures. N represents the number of times a component or "module" is replicated. Often one and the same N is used when replicating every component but we do not assume this. The *state machine approach* is a special case of N-modular redundancy in which the system is structured according to the *client/server* model and servers may be replicated [25]. The servers are the "state machines" in this approach. In this report we will focus on program transformations that implement the state machine approach. Although the name "state machine approach" is a poor one (perhaps "client/server" would be more descriptive), it is the one used in the literature and we will use it henceforth in

this report.

Many of the published algorithms for the state machine approach have been proved correct using mathematical reasoning. Using these algorithms in program transformations will increase the formality and precision of the KBSA, and thus will increase the confidence that can be placed in systems built using KBSA.

1.3 Project Overview

The Fault Tolerance in ADM project consisted of the following tasks and accomplishments.

- We extended the ADM specification language, *MSL*[1], with new constructs for fault tolerance. These constructs are needed so that fault tolerant applications can be programmed and fault tolerance properties of those applications can be specified in ADM. Because we are concerned with *distributed* fault tolerance, our language must also allow concurrent and distributed programs to be written. The ADM development team at Andersen does not plan to address the issue of fault tolerance specifications in the MSL design so we have addressed it ourselves on this project.

  This report presents our extension to MSL. We tried to keep the extension simple because language design is not our primary concern. At the same time we needed the extension to be powerful enough to allow most concurrent and distributed applications to be programmed easily, whether or not they are fault tolerant. The result is a general concurrent and distributed programming language with special declarations for fault tolerance properties.

  MSL is not yet complete. Therefore the MSL syntax we use in this report may not be exactly the syntax of the language at the end of the ADM project. Many important features of MSL, however, have been decided. For our purposes the most important feature of MSL is that it is object-oriented. Our language extension exploits the commonality between object-oriented languages and the client/server model so that the the constructs we added are appropriate both for MSL and for the state machine approach to fault tolerance.

- We implemented runtime support for the extended ADM specification language. A concurrent language needs more runtime support than does a sequential one because multiple threads of control must be maintained, and communication

[1]The language has recently been renamed "Argo" but we will use the earlier name in this report.

and synchronization must be possible between different threads. A distributed language may need even more runtime support than a concurrent one because network services must be provided. Fault tolerance applications expect still more from the runtime support, in particular communication facilities must be reliable, prompt, and must offer certain sequencing properties.

This report presents our design for the runtime support. Because our project has a limited budget we could not address all the runtime support issues in an ideal way. First, some of these issues are research topics that could easily consume more time than we have. Second, unusual network hardware and software may be the only way to get the proper runtime support for fault tolerant applications. We preferred not to spend project funds on non-standard hardware. We have taken an approach that concentrates on implementing the software runtime support for fault tolerance, while using only off-the-shelf hardware and existing operating system support where possible. The report evaluates the pros and cons of this approach and mentions some of the alternatives.

We have implemented a subset of the fault tolerance runtime support described in this report. The subset is sufficient to support many interesting fault tolerant applications, including an example we describe later. Our implementation is specific to C++, to Unix, and to TCP/IP networks.

- We implemented a fault tolerance program transformation. This transformation converts an application program into a fault tolerant distributed program with the same functionality. The original application program need not be written with any concern for fault tolerance other than some simple specifications of the desired fault tolerance properties. The fault tolerance properties of the application are specified using our extension to MSL. The transformation is automatic once the specifications are given.

  The transformation works by parsing an extended MSL source, converting it to C++. The C++ code generated by our tools will call the runtime support in a way that ensures fault tolerance.

  The transformation tools we implemented recognize only a subset of the extended MSL language. This subset is big enough to write interesting examples, yet leaves out parts of the language that would require us to write a much more sophisticated parser. The ADM project itself is developing the sophisticated parsing tools that are needed, so a future integration of our parser with the ADM parsing tools would implement fully the language described in this report.

  How does a programmer choose the fault tolerance specifications that should be embedded in an extended MSL program? These specifications come from

analyzing the system's reliability requirements. The general problem of requirements analysis for reliability is outside the scope of this effort.

- We investigated ways to improve the performance of the state machine approach to fault tolerance in specific application domains. In particular, we developed extensions to the state machine approach which can improve its performance when the servers manage large amounts of data.

- Using the tools developed on this project, we implemented an example fault tolerant application. The application is an extremely simplified air traffic control system. This system tracks aircraft and supplies information about the airspace traffic situation to its users, the air traffic controllers.

  The system works by maintaining a database of tracks representing the paths of aircraft. The basic functions of the system are:

  - to continuously collect new sensor data from radar and other sources;
  - to correlate new sensor data with known tracks;
  - to display the currently known aircraft tracks.

  Two clients and a single server implement this functionality. A more sophisticated system would undoubtedly use the track database to predict possible collisions and near-collisions and warn the air traffic controllers of this danger.

  Fault tolerance is clearly a desirable property for an air traffic control system because it can reduce the time the system is unavailable for monitoring the airspace. We wrote this example in our extended MSL, specifying that the system must tolerate a specified number of processor crashes, and we applied the fault tolerance transformation. The transformed system relies on the runtime support to ensure fault tolerance.

  Throughout this report we use examples to clarify the discussion. In most cases our examples are taken from this simple air traffic control application.

To summarize: we have designed and implemented a program transformation system for creating fault tolerant applications. The system involves an extension to the ADM language for specifying fault tolerance properties. It involves special transformation tools and runtime support. We have identified some theoretical problems needing to be solved to improve the runtime support, and we have solved some of them. We have implemented an interesting example application.

The tools are currently standalone, i.e., they do not depend on ADM to run. They are designed as extensions to ADM, however, and we anticipated that they might be fully integrated into ADM at some future time.

1.4 Report Outline

Chapter 2 summarizes the state machine approach to developing fault tolerant systems. This summary is based on the tutorial in [25] but we have tried to go beyond that reference in the following ways. First, we discuss the advantages of the state machine approach over the more traditional method of distributed fault tolerant design. Second, although we have not been as thorough as [25] in presenting the finer points of the approach, we have been more comprehensive in discussing all the cases that must be handled by the design of our runtime support. The finer points of the state machine approach are documented by the bibliographic citations.

Chapter 3 presents our extension to the ADM specification language for specifying fault tolerance properties and for programming concurrent and distributed algorithms. The chapter provides a rationale for some of the extended language constructs and a comparison with some other languages designed for concurrent and distributed programming.

The design of our program transformation and runtime support is explained in chapter 4. We introduce the basic concepts underlying the design. The design addresses questions of layering, including how runtime support facilities can be developed for concurrent and distributed programming as described in chapter 3, and which properties of network communication are assumed and which must be implemented. We identify the kinds of program information that the fault tolerance transformation must depend on. We superficially discuss our implementation of the tools and describe how these tools could be integrated more fully into the ADM environment.

Chapter 5 describes extensions to the state machine to improve performance and use resources more efficiently when servers manage large amounts of data. We call these extensions *Distributed Redundant Storage* (DRS). DRS stores data redundantly rather than simply replicating the data. This redundant storage approach is based on Redundant Arrays of Independent Disks (RAID) technology. DRS is essentially a distributed implementation of RAID. Chapter 5 includes a description of the communication protocols needed to implement RAID correctly in a distributed environment. DRS uses server storage more efficiently than the state machine approach, which is important when the servers manage large amounts of data, as in a large distributed database. DRS also uses processor resources more efficiently when servers are not failing. In addition, due to special properties of RAID–style redundancy, the DRS communication protocols can also decrease server latency. We describe these advantages in more detail in Chapter 5.

The report ends with chapter 6, which first reviews our accomplishments to date in

terms of the technical discussion in previous chapters. We then note the ways in which our implementation was incomplete. Finally, we list ways in which this work could be continued.

Chapter 2

The State Machine Approach

This chapter summarizes the state machine approach to developing fault tolerant systems. The state machine approach is a special case of *N-modular redundancy* and it is based on the *client/server* model for designing systems. We begin this chapter by explaining the ideas and terminology of N-modular redundancy and the client/server model because they will be used frequently in the rest of the report. Then we will show how the state machine approach adapts the client/server model for fault tolerance.

2.1 N-Modular Redundancy

N-modular redundancy is a method for reliable system design in which a system's components are replicated and the replicas are coordinated to tolerate failures. N is the number of times a component is replicated. N could be the same for all components, and often is, but in general N may be chosen separately for each component of the system.

A simple design for tolerating a single, arbitrary failure is to triplicate ($N = 3$) the component and to take a vote using the outputs from each replica. Then the failure of one replica will be invisible because its output will be outvoted, i.e., *masked.* Only after two replicas have failed will the triplicated component fail. If the probability of a replica failure is some small α, and if replica failures are independent, and if the probability of the voter's failure is negligible, then the probability of component failure is less than $3\alpha^2$. Because α is small, $3\alpha^2 < \alpha$, meaning that the triplicated component is less likely to fail and so is more reliable than the nonreplicated one.

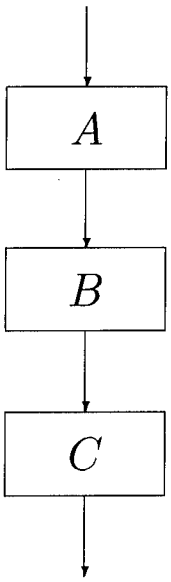


Figure 2.1: a sequence of computations on a single processor

This kind of probabilistic reasoning underlies every N-modular redundant design for any N.

N-modular redundancy was originally invented for hardware components but was readily adapted for systems with both hardware and software. Since the advent of computer networks N-modular redundancy has been used in the design of fault tolerant distributed systems. These designs replicate software components and distribute the replicas across a network.

N-modular designs differ in how the replica coordination is done. We show this by discussing an example.

In the triplicated system previously mentioned, the replicas could vote more often than once per output. Voting more often is an advantage for long-running or nonstop systems because an inconsistency between the state of the replicas will be detected, and possibly repaired, earlier. So imagine partitioning an arbitrary component's computation into a sequence of steps A, B, C, and so on. The computation might be depicted as in figure 2.1, where the arrows represent both the flow of control and the flow of data needed by the next step. Figure 2.2 depicts the same sequence of steps replicated on three processors, 1, 2, and 3. Voters, also replicated, precede steps B and C. Each step passes control and some data on to the next step. For example, B_1, the replica of step B running on processor 1, receives data from replicas A_1, A_2, and A_3. B_1 takes a majority vote of the data and uses the result of that vote as its

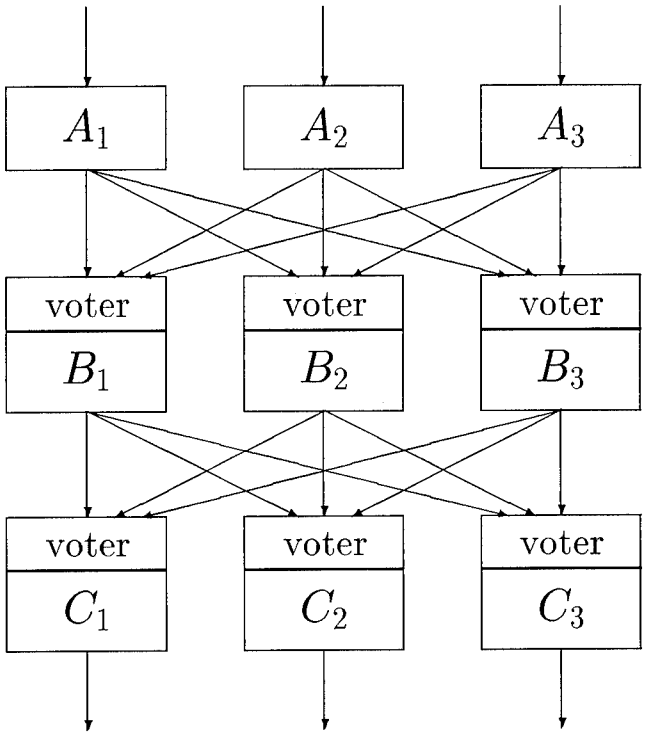


Figure 2.2: a sequence of replicated computations on processors 1, 2, and 3

input. If one of the A step replicas fails, the data it sends, or fails to send, will be outvoted.

We assume that each replica executes *deterministically* unless it fails. The output of a deterministic replica is a fixed function of its inputs and its starting state. With deterministic replicas it is certain that a majority vote will pick the correct output if fewer than half of the replicas have failed.

The three replicas must execute in synchrony: the same steps in the same order at roughly the same time. This synchrony is part of the replica coordination, and it is important. Although each step only needs to wait for 2 out of 3 of the preceding step's replicas to finish before a vote can determine the majority, the design must not allow one of the three replicas to fall far behind the others. Eventually the outputs from the slow replica would be lost and that would be equivalent to a failure of the replica.

This kind of N-modular design, in which replicas execute in synchronous lockstep and their outputs are voted, has been used frequently [35][15][32]. Several parameters of this design can be varied.

- The designer can choose the number of replicas. Increasing $N > 3$ tends to increase reliability but increases hardware costs and communication overhead, too. On the other hand, choosing $N = 2$ means majority voting will not mask an arbitrary failure. Instead some other means must be used to detect which replica has failed and to disable it. For example, if a replica is known to fail by simply halting then a vote is not needed: any output must be correct.

- The designer can choose how closely to synchronize the replicas. For example, a replicated computation can be synchronized after every processor clock cycle or instead after each computation step.

In spite of these options, the basic lockstep design still lacks the following kinds of flexibility.

- The designer may not want to replicate every component. Replication is expensive, both in hardware costs and in computation time. Perhaps the design should only replicate vital components while components of secondary importance are left unreplicated ($N = 1$). In that case the failure of an unreplicated component could give different data to vital replicated components, causing them to behave differently and eventually causing the system to fail. The design must prevent this possibility.

- The designer may want to interleave the steps of a computation differently on different processors, in order to increase the system's efficiency. Two steps are called *concurrent* if executing them in either order would have the same effect. The designer could take advantage of concurrency to compensate for variable delays in communication. For example, in figure 2.2, suppose that step B is concurrent with some other step D. Because of network delays the data sent from A_1 and A_2 may take a lot longer to reach B_3 than to reach B_1 or B_2. In that case it would be more efficient for processor 3 to execute step D while waiting for the data it needs from the other processors.

The state machine approach permits the flexibility just mentioned. The approach buys this flexibility by increasing the complexity of replica coordination beyond what is needed in the synchronous, lockstep, design. That complexity is the subject of this chapter.

To use the state machine approach, one must structure the system's computations according to the client/server model. We describe that model next.

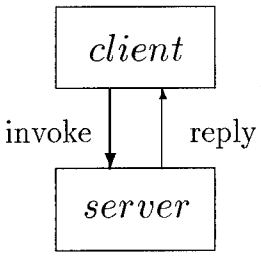


Figure 2.3: client invocation and server reply

2.2 Client/Server Model

In the client/server model, a system is structured as a collection of *services*. Each service is composed of one or more *commands*. These commands are *invoked* by *clients* in order to do useful work.

A service is implemented by a *server*. The sole function of a server is to receive commands and, depending on which command is received:

- to change the values of some, all, or none of its state variables;
- to compute a *reply* to the command if necessary.

Figure 2.3 shows a client invoking a command on a server and the server replying. The server state variables are internal, i.e., each variable can be accessed by exactly one server and none is shared between servers. The fact that state variables are internal makes it easier to determine when two servers can process commands concurrently.

Services may be *nested*, which means one service may depend on another. So if client C invokes a command on server S_1, then in order to carry out the command S_1 may need to invoke a command on server S_2. In that case S_1's service depends on S_2's. S_1's invocation on S_2 will be called a nested command. Note that in this case S_1 is both a server for client C and a client for server S_2.

2.3 Replicated Servers

In the state machine approach, a service is made fault tolerant by simultaneously executing several server *replicas*. Each replica identically implements the service's commands. Each replica has its own set of internal state variables, not shared with any other server or replica.

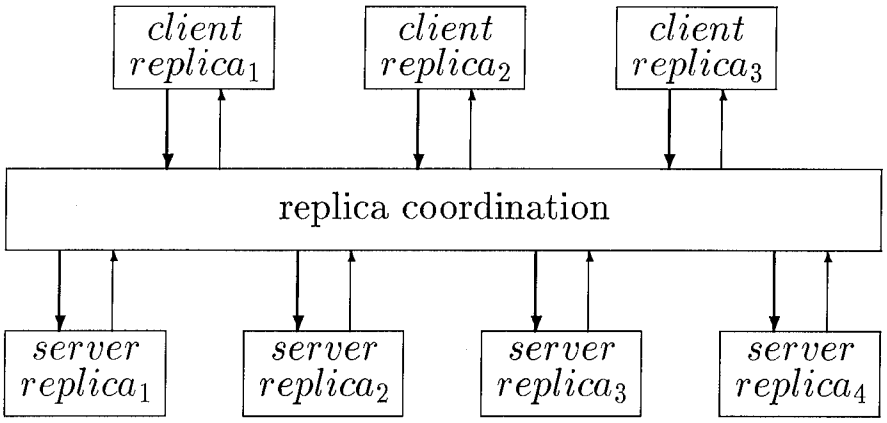


Figure 2.4: replicated client and replicated server

The replicas are coordinated so that if some of them fail, the service can continue to be provided correctly. Thus in the state machine approach the coordinated replicas act together as a single server, and we will continue to call the collection of replicas a "server", by analogy with the basic client/server model. We call a replica that has not yet failed *correct.*

A client may be replicated only if it is also a server. A replicated client may invoke commands on other servers, and these commands are, by definition, nested. A client and its server may be implemented by different numbers of replicas. The general situation is depicted in figure 2.4, which shows a triplicated client invoking a command on a quadruplicated server and receiving a reply. As in figure 2.3 the thick arrows represent the invocation and the thin arrows represent the reply, if there is one. The state machine approach handles all possible cases of replication vs. nonreplication: the client can be replicated, the server can be replicated, both, or neither. These special cases need to be kept in mind throughout the following discussion.

Replica coordination assumes the following property of replicas:

> All correct replicas of a given server will produce the same outputs if the same sequence of commands is invoked on each.

This assumption will hold for deterministic replicas. As a first approximation we will assume that all server replicas are deterministic, but we will handle the possibility of nondeterminism in section 2.9. Replica outputs include both replies to clients and nested commands to other servers.

Replica coordination guarantees the following property:

Every client will receive the correct reply to a command if enough server replicas are correct.

To implement this property, all correct replicas are given the same sequence of commands. Then correct replicas will produce the same replies because they are deterministic. Fault tolerance is achieved by combining the replies from correct replicas into a single correct reply, for example, by taking the majority vote of all replicas as previously depicted in figure 2.2.

To implement replica coordination one must solve the following problems.

- **Agreement** – Commands and replies can be lost or duplicated, and spurious ones can be generated by faulty components. The replica coordination must ensure that each correct replica processes the same set of commands and receives the same set of replies.
- **Order** – Commands can be delivered out of order. The replica coordination must ensure that each correct replica processes all commands in the same order.
- **Voting** – A single command can cause many replies, one copy from each correct replica and possibly others from failed replicas. Similarly, a replicated client sends many copies of the same command. Coordination must combine these multiple copies into a single one.

Agreement and order ensure that replicas process the same sequence of commands. Voting combines replicated inputs and outputs. Given a solution to these three problems, a server with enough correct replicas will always reply correctly to each of its clients.

In some designs, replica coordination also depends on solving one more problem.

- **Reconfiguration** – The number of replicas in a server may be changed for any of several reasons: replicas may fail, may be repaired and restarted, or may migrate from one network location to another. In any of these cases the server must be reconfigured in a way that preserves fault tolerance.

In later sections we will discuss how each of these four problems, agreement, order, voting, and reconfiguration can be solved. Every one of these problems has more than one solution, so we discuss the options available.

The replica coordination depicted in figure 2.4 is extra machinery that is added to both clients and servers. It complicates both clients and servers above and beyond the basic implementation and use of services.

What is the relationship between a computation in the state machine model of replicated clients and servers, and the sequence of computation steps described as an example of N-modular redundancy in section 2.1? The processing in clients and servers are the computation steps. Each command is an output from a client step and an input to a server step, while each reply is an output from a server step and an input to a client step. Section 2.1 claimed that the replicated client/server model offers more flexibility than the lockstep design example in that section, and we can now understand this flexibility more clearly:

- Not all steps must be replicated. Agreement and order are used for transmitting an output from an unreplicated step to a replicated one.
- The steps need not happen in the same order on every processor. Commands on two different servers are often concurrent and so the server replicas may process these commands in either order.

Thus the state machine approach is a useful way to think about flexible fault tolerant design.

2.4 Faults

The basic reason for replicating and distributing a server is to tolerate processor failures such as crashes. A fault in the processor hardware causes a failure that can in turn cause the failure of any or all server replicas running on the processor. In a good network design a failure of one processor does not imply the failure of any other processor, so those other processors can continue to run correct server replicas.

Usually the server replicas are exact copies of each other, each with the same structure of code and internal state variables. Copying and distributing replicas to multiple machines is adequate to tolerate processor failures. However, copying cannot tolerate design flaws in server code because the same flaw will be copied to every replica, causing all replicas to fail if any one does.

To tolerate design flaws, different replicas may be implemented differently, using different code and internal state variables, in the hope that a design flaw in one replica will be absent in others. In this case, called *design diversity* or *software fault tolerance* [2], the replicas may be run either on a single machine or on several.

Henceforth in this report we will assume that replicas are distributed only for the purpose of tolerating processor failures. We leave open the possibility of design diversity

but will not discuss any special support for it.

2.5 Failure Models

Solving the replica coordination problem depends crucially on what one assumes about the behavior of failed replicas and failed clients. These assumptions are called the *failure model.* Generally, the less is assumed (i.e., the weaker the model), the more elaborate the replica coordination must be.

The most commonly used failure models, from strongest to weakest, are defined as follows for component X:

- *correct:* X never fails.
- *fail-stop:* a failed X halts, but all other components have a mechanism guaranteed to detect that X has failed.
- *crash:* a failed X halts.
- *performance:* a failed X does not respond correctly to a sequence of inputs within a specific time limit.
- *authenticated Byzantine:* a failed X may behave in any manner except that it cannot corrupt messages it forwards from correct components and it cannot forge spurious messages that appear to come from correct components.
- *Byzantine:* a failed X may exhibit arbitrary behavior.

The failure models discussed in the rest of this report are crash, Byzantine, and correct.

Fault tolerance algorithms are characterized by how many failures of replicas having some failure model can be tolerated. For example, a system might tolerate up to 3 crashes of processors running replicas of a given server, but 4 such crashes could cause the server itself to fail. The greater the number of tolerable failures the greater the number of server replicas must be run. For example, if a server must handle commands from an unreplicated client and must tolerate up to t Byzantine processor failures then the server must have at least $3t + 1$ replicas [20].

In addition to the failure model for replicas we must consider failure models for the network that allows the replicas to communicate. Network failures can garble or

delay messages, introduce spurious messages, or, in the case of a network partition, prevent communication between some replicas completely. There are two ways to discuss these network failures:

1. A network can be treated as a special-purpose service whose components may fail. The network must be designed to tolerate a certain number of these component failures.

2. A failure of network communication between any pair of replicas can be treated as though it were a replica failure. For example, if the network is slow in delivering a message from replica A to replica B, the behavior of failed network plus correct replicas is indistinguishable from the behavior of a correct network plus a failed replica B that is slow in sending the message.

Every complete design for a fault tolerant distributed system must depend on some fault tolerance properties of the network. Therefore a complete design must ultimately use the first approach, in which network fault tolerance is a design problem that must be solved in addition to server fault tolerance. However, designing a network to ensure reliable replica-to-replica communication is not part of the state machine approach and we will need to know only a few of the details of reliable network design.

The second approach, treating network failures as though they were replica failures, is often convenient for discussing distributed services. We will use this approach occasionally in the report.

The rest of this chapter discusses various possible solutions to the problems of replica coordination. The reader may skip this discussion if desired, because later chapters do not depend on it.

2.6 Agreement

When a client invokes a command on a replicated server, one coordination requirement is to ensure that the command reaches every correct server replica and that all correct replicas agree which command was sent. This goal is called a *multicast*. *Agreement algorithms* have been designed to accomplish multicasting reliably even though network components and processors can fail.

A client has two basic strategies for multicast:

1. The client can send the command to every replica. This strategy is called *broadcasting*.

2. The client can send the command to every nearby replica and let those replicas, in turn, send the command to their neighbors. Only the first copy of a command is used, and later ones are ignored. This strategy is called *flooding* or *diffusion.*

The choice of strategy usually depends on the network topology: broadcasting is more natural when using a broadcast medium such as a bus, while flooding is appropriate for networks in which messages must be forwarded in order to reach some destinations.

When components can fail, the same problems arise in both strategies:

- The client can fail after having successfully sent the command to some replicas but not to others. To solve this problem, some replica or replicas must take over the client's task of transmitting the command to the other replicas.
- Replicas can fail. In the Byzantine failure models a failed replica can take over the client's task of transmitting but can transmit incorrect commands to other replicas. Byzantine agreement algorithms must ensure that such failed replicas are out-voted by the transmissions of correct replicas.

Many agreement algorithms have been published and studied. An example of an agreement algorithm using the broadcasting strategy can be found in [9], while an example using the flooding strategy can be found in [23]. Algorithms for Byzantine agreement are discussed in [20] and [4] gives a survey.

Reliable multicasting using agreement algorithms may also be needed when a server replica replies to a replicated client. The same problems arise in this case.

2.7 Order

Even though replicas can be made to receive and to agree about every command, commands may arrive at different replicas in different orders. These different orders result from delays in network transmission. Another coordination requirement is to ensure that all correct replicas process commands in the same order regardless of network delays.

Putting the commands from a single client in order is easy: the client simply tags each command in a sequence with a count of its position in the sequence. It is more difficult to put the commands from all clients in the same order at all replicas. For example, if client C invokes the sequence of commands c_1, c_2 and client D invokes commands d_1, d_2, server replica R may receive c_1, d_1, c_2, d_2 while replica S receives

c_1, d_1, d_2, c_2. Then the command sequences differ at the replicas even though each sequence preserves the order of commands from each client.

The basic idea for ordering commands is to create a unique global timestamp for each one and to process smaller timestamps first. Two approaches exist:

1. Timestamp using the time obtained from a synchronized distributed clock.
2. Timestamp using a *logical clock* [17] value that respects potential causality, i.e., if command or reply x could potentially influence command or reply y then x's timestamp is smaller than y's.

In both approaches the clock must be maintained at all replicas and all their potential clients.

A command is called *stable* once it is certain that no command with a smaller timestamp can be received later. A replica must not process a command until it is stable. To determine stability when using real time clocks one must age each command longer than the longest possible network delay; this method relies on the synchronization of clocks at replicas and clients. To determine stability when using logical clocks one waits until, for every potential source of commands, either a later command has been received or the source has failed. Algorithms for logical clock stability differ in several ways, for example, whether the client or replicas issue the timestamps for commands [25].

An example of ordering using real time can be found in [8], while a system based on logical clock timestamps is described in [5].

Note that the state machine approach does not place any requirement on the order of replies received by client replicas. This ordering is usually done within the client itself, by matching each reply with the command that caused it. In some applications the client wants to keep the flexibility of receiving replies in any order.

2.8 Voting

A client invoking a command on a replicated server will receive several replies and must vote to determine which reply to use. The failure model for the replicas will determine what kind of voting the client must use.

- The client uses majority voting if replicas are Byzantine and can generate spurious replies.

- The client takes the first reply if replicas can suffer crash failures; all later replies to the same command are ignored.

Server replicas also need to use voting to combine commands from client replicas. When a client replicated n times invokes a command on a server replicated m times, $m \times n$ copies of the command will be sent from client to server. Each server replica combines the command copies using an appropriate method of voting, as already discussed for replies. The number of command copies that need to be transmitted in this case can be minimized by using a digital signature technique [12].

Note that voting makes running an agreement algorithm unnecessary. If all replicas that send a message are deterministic, each receiving replica will always receive enough correct, identical copies of the message to outvote missing or corrupted copies. This reasoning applies both to client replicas sending commands and to server replicas sending replies.

Agreement is used in place of voting only when there are so few sender replicas that failed replicas will not always be outvoted. In particular, agreement is necessary when an unreplicated client invokes a command on a replicated server and when an unreplicated server replies to a replicated client.

2.9 Nondeterminism

Correct server replicas will produce the same outputs if they are given the same sequence of commands and if they execute commands *deterministically.* A deterministic replica produces a set of outputs and changes its state in a way that is purely a function of the command and the previous state.

We have so far assumed that servers must be deterministic. This assumption rules out replicating some servers, for example, those that depend on a random number generator, or on the real time, or on values from replicated sensors. If an application must use nondeterminism in an essential way then its designer can take either of two approaches:

1. The application can be designed so that the nondeterminism is localized in clients, not in servers. This restriction on servers poses no problem in principle, though in practice it may be a significant inconvenience for the designer.

2. A server implemented by a collection of nondeterministic server replicas can use an agreement algorithm to resolve the differences in the replicas' outputs

and state changes before interacting with its client or other servers. This strategy for replica coordination is more difficult to implement than coordinating deterministic replicas. We explain this strategy further below.

In either case all correct server replicas will give the same outputs in response to the same commands, in the first case because of replica determinism and in the second case because of special coordination between nondeterministic replicas.

In a server with nondeterministic replicas, each replica computes a candidate output. Before sending the candidate output to the client or to other servers, each replica transmits its candidate output to every other replica in the same server, using an agreement algorithm. The replicas then use some prearranged method for combining the outputs of all replicas into a single output, e.g., taking an average. All correct replicas then adopt the single output as their own.

This strategy for handling nondeterminism gets more complicated if a command causes not only output but internal state changes. Then all correct replicas must agree on how to change their state. The method they use is highly dependent on the internal details of the replicated server.

A server that takes input from a physical sensor is the most common case in which replicas will behave nondeterministically. If the server replicas each read the same physical sensor, they will read it at slightly different times and so may get different readings. If the sensor itself is replicated for fault tolerance, then each server replica will read one sensor replica and again, different readings may be obtained. Sensor-reading replicas must use an agreement algorithm to get consistent sensor data. Figure 2.5 depicts four server replicas reading four sensor replicas. The sensor values are shown, and in one case the sensor value is missing. The agreement algorithm allows the replicas to return a unified value from the sensors.

2.10 Reconfiguration

Reconfiguration in the state machine approach means adding or subtracting replicas from a server. The most common reason for reconfiguration is that the processor on which a replica was running fails, is removed from the system temporarily, is repaired, and is reintroduced into the system. In long-running systems reconfiguration may be needed also for preventive maintenance.

If the design of a server uses reconfiguration, then the following problems must be solved:

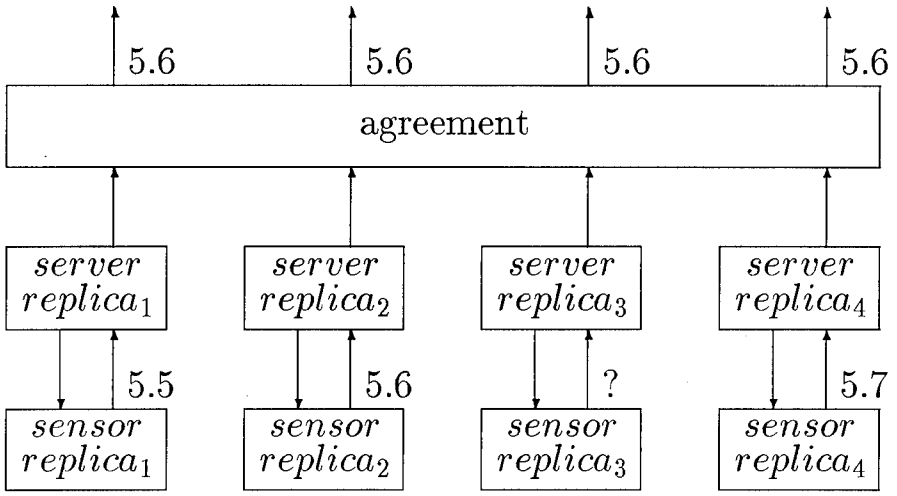


Figure 2.5: nondeterministic input from sensors

- A replica just starting up must have its internal state brought up to date. There are two basic methods for doing this. Either the new replica can copy its state from some other up-to-date replica, or it can be sent a history of commands given to other replicas and process them one at a time. Both of these methods need special-purpose machinery to be built into state machine replicas.
- Every client or replica that uses a list of the server replicas (for multicasting, say) must have its replica list updated. If these lists are distributed to clients dynamically, maintaining the lists by notifying all clients of changes will be difficult. Updating the list of server replicas is easier if the list is centralized in, say, a special fault tolerant service that maps the name of a service into its current list of replicas.

Reconfiguration is not needed in some systems. In these systems, the probability is small that enough processors will fail to cause servers to fail. Then the set of replicas in each server may be fixed when a system is started.

2.11 Summary

In the state machine approach to fault tolerant design, a system is structured into a collection of services. Any server may be replicated, and each server's replicas are coordinated. The state machine approach allows great flexibility in how to coordinate

replicas, but problems of agreement, order, voting, and reconfiguration must be solved as part of the design.

Chapter 3

Specification Language

In this chapter we describe a language for specifying and programming fault tolerant software. The chapter begins by describing some features of the ADM specification language we need. Then we extend the ADM language with new language constructs for expressing concurrency, distributivity, and fault tolerance properties. Even though the design of the ADM specification language is not yet finished, enough of its features have been settled that our extensions to it should not need to be significantly modified when its design is complete.

3.1 The ADM Specification Language

Users of the KBSA Advanced Development Model (ADM) will write programs and program specifications in a new language created to support KBSA goals. This new language is currently under development at Andersen Consulting and is called MSL, for Minimum Specification Language[1].

Even though MSL is not yet complete its outline is already clear enough that we can design extensions to it for writing fault tolerant programs. These extensions must allow one to write programs that are explicitly concurrent and distributed, and in which fault tolerance properties can be specified. Our extensions to MSL do not conflict with the language design work at Andersen Consulting because it has been decided that MSL will have no explicit support for either concurrency or distributivity.

MSL has the following basic features:

[1]The name has recently been changed to Argo, but we use the older name in this report.

- It is object-oriented, with class declarations based on a recent object-oriented database management standard [7].
- Its syntax resembles that of the C++ programming language [29].
- It permits assertions about program state to be embedded in sequential code as preconditions, postconditions, and invariants [13][21].
- It allows encapsulation of closely related classes into modules.

Generally we do not assume that the reader of this section is familiar with C++, embedded assertions, or approaches to modularity. We do assume some familiarity with object-oriented concepts, though. We will summarize the basic object-oriented terminology in the next section.

3.1.1 MSL Syntax

To describe our language extensions we need to know only a small subset of MSL. Our discussion of this part reflects the state of MSL in early 1995. At that time, MSL was divided into two parts: declarative and definitional. The declarative part had received the most attention from Andersen and was nearly complete. The definitional part was simply C++. The syntax we present in this section reflects the syntax of the declarative part exactly. For the definitional part, however, we have simplified the C++ syntax slightly in a way that seemed consistent with Andersen's longer-term goals for MSL[2].

Both MSL and C++ are object-oriented languages, which means that a program's code and data can be structured into units called *classes*. A class is a set of *objects* that a program can potentially create, along with operations that the program can use to modify any of those objects. The class operations are called *methods*[3].

To describe the syntax of a programming language we will use examples of code taken from a hypothetical air traffic control *database*. This database comprises a set of aircraft *tracks*.

[2]We have simplified the C++ syntax by removing the "*" operator. We have changed the semantics accordingly so that every variable that refers to an object of a user-defined class is implicitly a pointer. Every variable that refers to an object of a predefined class, such as "int", is simply a name for the object itself, as in C++.

[3]In C++ a method is really called a "member function", but "method" is the traditional term and we stick with it here.

Given a class `track`, the syntax `track t;` declares that variable `t` names (i.e., is currently a pointer to) a `track` object. This declaration doesn't create any `track` object, though. The syntax `new track` is an expression that allocates memory on the heap for an object of class `track` and returns a pointer to the new object. Combining the declaration of a variable with the expression to create an object we have the syntax

```
track t = new track;
```

that creates a new object and initializes `t` to name it. Alternatively, we could declare the variable and initialize it separately with an assignment statement:

```
track t;
t = new track;
```

To declare a function, named `coalesce`, for creating a new track from a set of points in the airspace, one might write

```
track coalesce (point_set p, distance max);
```

This function returns an object of class `track` and is parameterized by two arguments `p` and `max`, which are themselves objects of classes `point_set` and `distance` respectively. To call the function one might write

```
coalesce (collection, goodness_of_fit+delta)
```

which is an expression that first causes the two subexpressions `collection` and `goodness_of_fit+delta` to be evaluated, then applies function `coalesce` to the resulting objects, and finally returns the `track` value computed by `coalesce`.

An example declaration for a `database` class could be written

```
class database {
  public:
    attribute boolean stale;
    relation track_set data;
    error update ();
    vector closest_approach (track t, distance uncertainty);
  private:
    relation date last_updated;
}
```

This declaration says that a `database` has three state variables, `last_updated`, `stale`, and `data`, of classes `date`, `boolean`, and `track_set` respectively. These variables are indicated by the `relation` and `attribute` keywords[4]. It also has two methods, named `update` and `closest_approach`. The latter method, for example, takes two arguments of classes `track` and `distance` and returns a `vector`. The keyword `public` means that the names "stale", "data", and "closest_approach" can be used to refer to these variables and methods in parts of the program outside this class definition, while the keyword `private` means that the name "last_updated" can only be used within the class.

Note that methods are declared using exactly the same syntax as functions. The difference is that a method is declared within the class definition and thus takes on a different meaning: a method is a function that takes an object of the class, in addition to the normal function arguments, and will generally affect the state of that object in addition to returning the normal function result value. In the previous example, both `update` and `closest_approach` are applied to a particular `database` and may change the state of that `database`.

The syntax for calling a method differs from the syntax for calling a function. For example, if `db` is a `database`,

```
db->closest_approach (tr[i], delta)
```

calls the method `closest_approach` for object `db` after evaluating expressions `tr[i]` and `delta`.

A similar syntax is used to access state variables in an object. For example

```
db->stale = true;
```

assigns the boolean value `true` to the state variable `stale` in the `database` object `db`.

Object-oriented languages always provide for one class to *inherit* features of another class. We will not need to know the mechanics of inheritance in MSL, however.

We will need to use MSL arrays. An array of objects, like a single object, is allocated with `new` but the size of the array is indicated in square brackets. For example,

```
track tr = new track [5];
```

[4]A relation differs from attribute in that the former takes on values from a user-defined class and may have an inverse. However, we did not enforce this distinction.

allocates space on the heap for 5 objects of class `track`, and initializes variable `tr` to name the array. MSL expressions that refer to the 5 objects are

```
tr[0], tr[1], ..., tr[4]
```

Note that as in C++, there is no syntactic distinction between a variable that names an object and one that names an array of objects: the declaration `track tr` could serve either purpose.

We will not need to know the syntax of MSL expressions in greater detail than already described.

3.2 Extensions for Fault Tolerance

We extend MSL for concurrency, distributivity, and fault tolerance. The result is an MSL superset called Concurrent MSL (CMSL). This superset allows for servers to be started and for clients to invoke server commands. It allows the fault tolerance properties of servers to be specified by declarations, and these declarations also allow the programmer some control over how servers and server replicas are distributed to different processors on the network. CMSL is intended to be useful not only for fault tolerance but also for most concurrent programming applications.

3.3 Concurrent MSL

An object-oriented language such as MSL provides a natural way to describe services: a service is a class of objects and each of the service's commands is a method available in that class. So a programmer using MSL is already using the client/server model. Every object the program allocates is, in effect, a server. Every time the program calls a method, an invocation happens, and if the method returns a value, a reply happens. The correspondence between object-oriented constructs and the client/server model is a close one, and it is shown in figure 3.1.

We use the correspondence between object-oriented constructs and the client/server model as the basis for CMSL. This correspondence is no coincidence: it derives from the need to structure a program into self-contained units with explicit interfaces. The correspondence breaks down when the units are not self-contained. For example, MSL allows objects to share variables, but the client/server model prohibits this. We

client/server	object-oriented
service	class
server	object
command	method
client	caller
invocation	call
reply	return

Figure 3.1: correspondence between client/server and object-oriented

will note other exceptions to the basic correspondence as we present the details of CMSL.

Do MSL objects run in parallel on different processors, as one might expect if they are acting as independent servers? Most likely an MSL program runs on one processor, but not necessarily. The MSL programmer need not know the answer to this question because the MSL compiler chooses for him whether an object is an ordinary programming object or a concurrently running server, and whether such a server is local or remote. The meaning of the program will be the same no matter which choice the compiler makes. Are MSL objects replicated and fault tolerant? Similarly, the answer to this question does not matter: the meaning of an MSL program is its meaning when run on a single processor, and if that processor fails, all MSL objects will fail with it. So MSL can be thought of as already *implicitly* concurrent and fault tolerant.

Implicit concurrency, though, is not good enough for our needs. Present-day compilers are not sophisticated enough to choose, in most cases, which program objects should be made concurrently running servers for the sake of program efficiency. Solving this program design problem automatically is beyond the state of the art for compilers. The next sections show how concurrency is instead specified *explicitly* in CMSL.

CMSL adds only a few constructs to MSL.

- Section 3.3.1 introduces a construct for creating new servers. Both the functionality and fault tolerance properties of each server are specified using this construct.

  Note that CMSL currently provides no construct for destroying servers. Such a construct is unimportant in a language used only for experimental programming but would need to be added to make CMSL programming "industrial strength".

- Section 3.3.2 adds a construct for remote procedure calls. These calls combine

both an invocation and its reply in a single program step. Remote procedure calls make programming in the client/server model look very similar to ordinary sequential programming.

- Section 3.3.3 adds separate constructs for invocations and for replies. Using them, a programmer can invoke a command but do other computation while waiting for the reply. This separation of functions allows distributed programming to be made more efficient.

Section 3.3.4 concludes the discussion of CMSL by summarizing the syntax of the new language constructs.

3.3.1 Parallelism

Because of the close correspondence between the client/server model and object-oriented constructs, we choose MSL objects as the unit of parallelism in CMSL.

This choice is not the only choice. For example, Ada-83 [33] uses its modularity construct to define classes, but it provides a separate construct for parallelism. In contrast, NIL [28] uses the same construct for both modularity and parallelism, but not object-oriented classes. We made our choice to minimize the differences between programming in MSL and in CMSL. We believe that this kind of transparency is consonant with the project's goal of allowing fault tolerance to be programmed transparently.

In CMSL, every server is an object of a server class. Thus the declarative part of CMSL must allow one to specify which classes are server classes. We add the `server` keyword for this purpose. For example:

```
server class database { ... }
```

declares a server class named `database`.

One creates a server (i.e., an instance of a server class) with the following expression:

`server` *class optional-size optional-properties*

The *class* is the name of an MSL class whose methods define the server's commands and whose variables define the server's internal state. Every server executes its methods when they are called, but each method is executed completely before the next one

is begun. The *optional-size* declares that an array of servers is to be created; the size of the array is specified by a positive integer expression in square brackets. If the size is omitted, exactly one server is created. The *optional-properties* is a list of properties to be shared by each server created; we list the possible property specifications later.

For example, the expression

```
server database [3]
```

creates 3 servers that implement the `database` class.

A CMSL expression using the `server` keyword is analogous to an MSL expression using `new`. Both return a pointer to an object or an array of a given class. While `new` points to an object on the heap, though, `server` points to some representation of an object that will be run concurrently and may be distributed across the network. So the declaration

```
database serv = server database [3];
```

initializes the variable `serv` to name the database array. The databases are named `serv[0]` through `serv[2]`. The analogous MSL declaration

```
database serv = new database [3];
```

also creates 3 databases, but they are all local objects.

As part of the `server` expression zero or more of the following properties may be specified, in any order.

- `priority` (*integer*) specifies that the server is to be run with the *integer* priority. If the priority is not specified, a default priority of `0` is used.
- `replicas` (*positive integer*) specifies that the server is to be replicated the *positive integer* number of times unless more replicas are needed to satisfy the tolerance and failure model specifications. The default is `1`.
- `tolerance` (*nonnegative integer*) specifies that the server must tolerate the *nonnegative integer* number of replica failures. The default is `0`.
- `model` *failure-model* specifies that any failed server replicas are assumed to behave according to the specified *failure-model*, which is one of the following:

- `correct`
- `crash`
- `byzantine`

The default is `correct`.

- `hosts` (*host*, *host*, ..., *host*) specifies that the server is to be run preferably on the *host*s named, and a host named earlier is preferred over one named later. If the server is replicated, each replica is to be run preferably on one of the named hosts, but in no case will two replicas be run on the same host, whether named or not.

  A `hosts` specification does not commit the CMSL runtime system to choose the particular hosts named because it may be impossible for some reason to use one of the named hosts; the named hosts will be used if possible. If no hosts are specified, or fewer are specified than are needed for all server replicas, then the CMSL runtime system may freely choose the unspecified hosts.

- `disjoint` specifies that each replica of each server in an array created with a single `server` expression will be run on a different host, if possible. If not possible, then every pair of servers is run either on the same set of hosts or on disjoint sets. This keyword has no effect if only a single server is being created. When `disjoint` is not specified the replicas of each server in an array are allocated to the same set of hosts, according to the `hosts` specifier.

  The fewer the hosts used by an array of servers the more reliable is the array. However, if for performance an application uses a given number of hosts anyway, then the entire application is most reliable if its servers use either the same or disjoint sets of hosts. This fact is proved in [22] for Byzantine fault tolerant servers each with the same number of replicas. We conjecture it holds also for crash fault tolerant servers but we have not proved that it does. Thus even though the `disjoint` specifier does not allow fine-grained control over allocation of replicas to hosts, it does allow one to choose two of the most useful cases.

As an example, the following declaration initializes variable `serv` to refer to a newly-created server of class `database`, unreplicated, with priority 5:

```
database serv = server database priority (5);
```

The next expression creates a server with 4 replicas to tolerate a single Byzantine failure. Two of the replicas should be run on hosts `h1` and `h2`, while the other two

will be run on two other hosts. Note that the specification of 3 replicas is overridden because in general 4 replicas will be needed to tolerate a single Byzantine failure.

```
server database model byzantine tolerate (1) replicas (3) hosts (h1,h2)
```

Why not reject the previous declaration at compile-time, because it specifies an impossible number of replicas? A compile-time check is possible in this case but it is not required, because it is not always possible: the number of faults tolerated and the number of replicas are expressions to be evaluated at run-time, so their values are not always known at compile-time. Also note that the number of replicas may be more than the minimum needed, as in the following example.

```
server database model byzantine tolerate (1) replicas (5)
```

The next expression creates an array of 3 server each with the 2 replicas needed to tolerate 1 crash failure, and if possible the replicas are run on 6 different hosts.

```
server database [3] model crash tolerate (1) disjoint
```

Hosts

The class `host` is predefined in CMSL. Objects of this class represent actual, physical, processors in the network.

Because different network protocols have different ways of assigning addresses to processors, certain features of the `host` class must be implementation-dependent. For example, if CMSL is implemented on a network using the Internet Protocol (IP), the IP address should be supplied when creating a new `host` object. We will not discuss these implementation-dependent details of the `host` class further.

A few machine-independent operations are possible for `hosts`. For example, `host` variables may be compared for equality, or one may be assigned to another[5]. Also,

```
host my_host ();
```

is the declaration of a predefined function of no arguments that returns the local `host`.

[5]Assignment copies the pointer to the object, not the object itself. Similarly, comparing for equality means comparing pointers, not the values of the objects' internal state variables.

Shared State Variables

Not every MSL class can be named in a `server` expression in CMSL. This constraint comes from a basic idea in the client/server model: servers do not share state variables with other servers or with their clients. Therefore, because we want to use class `database` to define `servers`, no method in `database` may access a variable declared outside the class.

MSL's rules on which variables are visible (i.e., accessible) at a given point in a program have not been fully worked out. It is likely, though, that MSL will allow methods to access variables that are global to the program that starts `database`. CMSL's constraint on shared variables in a class is likely, therefore, to be stricter than MSL's.

Some languages for concurrent programming, such as Linda [6], permit variables to be shared between distributed processes. Because shared variables are very useful in programming, the abstraction of distributed shared variables can simplify writing applications. This abstraction, though, has its costs. It is difficult to implement for arbitrary shared variables in a CMSL program, and it is built from more basic kinds of interprocess communication. Both these facts suggest that the compiler for CMSL would need to be made more complicated to handle distributed shared variables.

In designing CMSL we thought that communication through shared variables was much less important, more costly to implement, and less in harmony with the underlying object-oriented philosophy than communication in which a client calls a server's methods. We now turn our attention to the latter kind of communication.

3.3.2 Synchronous Communication

Using the `server` construct, a CMSL programmer can write programs that consist of several servers and their clients executing concurrently. We must also introduce constructs to control interactions between these concurrent entities.

CMSL constructs are of two kinds: synchronous and asynchronous. A construct is *synchronous* if completing it involves waiting for a response from another concurrently running server. A construct is *asynchronous* if it need not wait. The program constructs in each client and server are executed sequentially just as in any MSL program, but asynchronous constructs are ones that can go on to the next construct while leaving an interaction unfinished.

Starting a server is synchronous in CMSL. The runtime support system must guar-

antee that the new server has been started and is ready to receive commands before the next programming language construct is executed. This guarantee is crucial if the next step in the program happens to invoke a command on the newly-created server; otherwise the invocation might reach the server before the server was ready to receive it, causing the program to behave unpredictably.

After a server is started, the simplest way to communicate with it is to *call* it. A call, sometimes referred to as a "remote procedure call", is analogous to calling a method in MSL. The client tells the server which method to invoke and what arguments to use. If the method returns a value, the client waits for that value from the server – it is this waiting that makes calling a synchronous construct.

A CMSL call on a server's command is written exactly like a call on an object's method in MSL. Using our previous example in which `serv` names a server in class `database` and `update` names a method in that class, then

```
serv->update ()
```

is the expression for calling that method.

A server that receives an invocation will execute the corresponding method until it is complete. These semantics for invocations match the semantics in MSL and in the client/server model.

Though calls in MSL and CMSL look the same, their meaning must be different [3]. The differences between calling a server in CMSL and calling a method for an ordinary MSL object are:

- *performance:* the remote procedure call usually takes more clock time than calling a method because it usually depends on network communication that is relatively slow.
- *failures:* the remote procedure call can fail for any of the reasons that calling a method can, but it can also fail if the server is unavailable for any reason.
- *parameter passing:* CMSL uses call-by-value instead of call-by-reference semantics for objects of user-defined classes passed as arguments in a remote procedure call, and constraints are placed on the objects that can be passed.

The performance cost of a remote call can be significant. The actual cost depends on many factors, including the operating system, processor speed, and network type.

If the server fails during a call, CMSL aborts the call and raises an exception in the caller.

If server failure is a significant risk, the CMSL programmer uses the fault tolerance specifications described in section 3.3.1 to minimize that risk. These specifications cause fault tolerance mechanisms to be built into the server automatically. Yet even a fault tolerant server can fail if enough of its replicas fail. CMSL contains no other mechanisms for recovery from server failure, such as the checkpointing used in Aeolus [36]; the reconfiguration mechanisms used in the state machine approach supersede such checkpointing mechanisms.

Parameter passing in the fragment of MSL we have discussed has call-by-value semantics for predefined classes and call-by-reference semantics for user-defined classes. Call-by-reference means that an argument can be modified and returned in its modified condition; call-by-value means that an argument is used for its value only and cannot be modified.

Call-by-reference is very hard to implement for remote procedure calls [31]. If an object is evaluated as an argument, sent to a remote server, modified, and returned, it is not clear in general how the modified object will be used to update objects on the local host. For example, suppose one argument is a linked list that is passed by reference and some elements are deleted from the list on the remote server. When the altered list is returned, it cannot simply replace the old list because there may be pointers to the old list that would be left dangling. Another difficulty arises if the same object is passed via two different function arguments and those arguments are updated by a remote server. How will the implementation ensure that the updates are made to the same object? CMSL will sidestep these problems by passing *all* objects by value when calling a server.

Also, argument values that contain circular references, – e.g., object A points to object B, which points to object A – are disallowed for simplicity in implementing the mechanism for copying values from a client to a server, and vice versa. It is not necessary for the CMSL compiler to check for such circularity, but if it does not, a circular argument value could cause an infinite loop in evaluating the argument.

An array may be passed as an argument in a CMSL remote procedure call. The length of an array argument is determined from its declaration, e.g.,

```
track tr[5];
vector vect = db->closest_approach (tr, 19);
```

passes the entire `track` array, while

```
vect = db->closest_approach (tr[2], 19);
```

passes only a single track, the third array element.

3.3.3 Asynchronous Communication

A remote procedure call that must wait for a return value will often throw away opportunities for increasing a program's efficiency. If the program could wait for several calls at once, collecting the return values as they come in, it could exploit the fact that servers running independently on different remote hosts can process the calls in parallel. To use the client/server terminology, a client does the following when making 3 calls in a row:

invoke host 1
wait for reply from host 1
invoke host 2
wait for reply from host 2
invoke host 3
wait for reply from host 3

but would prefer to do this for efficiency:

invoke host 1
invoke host 2
invoke host 3
wait for any reply from hosts 1, 2, or 3
wait for any reply from hosts 1, 2, or 3
wait for any reply from hosts 1, 2, or 3

Asynchronous communication makes the second approach possible.

Futures

To build a remote procedure call using asynchronous communication, the language must provide a mechanism to relate each reply that is received to the invocation that caused it. This mechanism is called a *future*. Each remote procedure call is associated with a unique future. Each invocation produces the future, and each reply carries with it the knowledge of the future it is a reply for. The future is a kind of ticket for picking up the reply later.

The class `future` is predefined in CMSL. The programmer manipulates an `future`

using the ordinary assignment and equality operations, and also by querying its result. For a future declared by `future f` the latter operation would be written `f->result`. For example,

```
future f;
vector v;
invoke a command that returns a vector value
wait for reply to future f
v = f->result;
```

We expect that type checking in CMSL, and in MSL, will be done when a program is compiled. Then if a statement involving an `future`, such as `v=f->result;` is to be type-checked, the compiler must be able to determine that `v` and `f->result` are objects of the same `vector` class. So there must be a different `future` class for every class of object that can be returned in the program. In C++ one can create a class definition that has another class as a parameter, using a construct called a template, or generic. MSL will have such a generic class construct, and that construct could be used for CMSL actions. In the rest of this report, however, we will ignore the question of type-checking `future` expressions in CMSL.

Invocations

To invoke a command on a server the CMSL programmer uses the same syntax as used for remote procedure call, but precedes it with the keyword `invoke`. An invocation returns a future. With this new syntax we can fill in more of the previous example:

```
future f;
vector v;
database db = server database;
track tr;
distance d;
f = invoke db->closest_approach (tr, d);
wait for reply to future f
v = f->result;
```

The invocation does not wait for a reply, but returns the future immediately.

The constraints placed on parameters to calls in section 3.3.2 apply to invocation parameters also.

Replies

The replies to several concurrent invocations can be received in any order because of network communication delays and because of nondeterministic choices made by the runtime support system. This fact complicates the design of a language construct to receive replies. A further complication is that a server, even a fault tolerant server, can fail, in which case its reply will never be received and the invoking program might wait forever.

The CMSL `accept` construct lets the programmer choose, nondeterministically, the next available reply in a given set. It also permits a timeout clause in case no reply is received within a specified time. The timeout clause gives the programmer a crude mechanism to detect that a server has failed and to proceed anyway.

`accept` (*future-array, positive-integer*) *optional-timeout*

The *future-array* is an array of `futures` of the same class. *positive-integer* specifies how many elements of the array are to be waited on. We will discuss *optional-timeout* later. The `accept` expression returns a positive integer that is the index into the *future-array* for the reply that is received next.

With the syntax for accepting replies we can now complete the previous example:

```
future f;
vector v;
database db = server database;
track tr;
distance d;
f = invoke db->closest_approach (tr, d);
accept (f, 1);
v = f->result;
```

The array index returned by `accept` is ignored in this case because the array of futures has only one element.

As another example involving arrays of `futures` consider this program fragment:

```
database db = server database [5] disjoint;
future fut [5];
track tr;
distance d;
```

```
vector vect;
int n, index;
```
for every n, $0 <= n < 5$,

```
    fut [n] = invoke db[n]->closest_approach (tr, d);
```

for every n, $0 <= n < 5$,

```
    index = accept (fut, 5);
    vect = fut [index]->result;
```

process the resulting vector for case index

In this example, five independent **database** servers are created and five commands are invoked, one on each server, to be processed in parallel. The results are received and processed in any order. Note that the results might be handled very differently for each of the five servers, which is why **accept** returns an index into the **future** array rather than the future itself: the MSL case-split construct can be used to separate the three cases based on which index was returned. The index must refer to one of the futures being accepted, of course, and in this case, $0 <=$ **index** < 5.

If the programmer wants to allow the possibility of timing out when no reply is received, then the *optional-timeout* clause can be included. This clause is written

timeout (*nonnegative-real*) *timeout-code*

where *nonnegative-real* expresses the minimum amount of time, in seconds, that the client will wait for a reply from any **future** named in an **accept**. The expression has any real number type available in MSL[6]. If no reply arrives within that time, an exception is raised. For example,

```
future fut;
int index;
index = accept (fut, 1) timeout (5.0);
```

In this program fragment the **accept** waits on just one **future**. If it doesn't arrive within **5.0** seconds then an exception is raised and the value of **index** is undefined.

Now that both **invoke** and **accept** have been introduced into CMSL, we can see that remote procedure call can easily be implemented using them. For example the call

[6]In C++ the types "float" and "double" could be used here.

```
error err;
err = serv->update ();
```

means that the method update is invoked and the caller waits for the value err to be returned. This call could be replaced by

```
error err;
future fut;
int index;
fut = invoke serv->update ();
index = accept (fut, 1) timeout (10.0);
err = fut->result;
```

The actual implementation of remote procedure call will probably be more sophisticated in choosing the length of the timeout based on knowledge of the application and its environment.

Nondeterminism

As explained in section 2.9, a nondeterministic replicated server presents special problems for replica coordination. CMSL currently provides no support for coordinating nondeterministic replicas.

Can the CMSL compiler check that a server is deterministic? No compiler check can decide whether a server is deterministic in every case, but a check could be implemented that recognizes determinism in certain cases. One reason checking determinism is hard is that a programmer can use asynchronous invocations, handle the replies in a nondeterministic order, yet ensure that the service being programmed is deterministic regardless of the order of replies. For this reason CMSL makes it entirely the programmer's responsibility to ensure that a server is deterministic.

3.3.4 Concurrent MSL Summary

The extensions to MSL syntax for concurrency are few. We introduced constructs for

- starting a server;
- invoking a server command;

- receiving a server reply.

No change to the syntax was needed for remote procedure calls, although the semantics of these calls differs from ordinary calls on MSL methods.

CMSL has these additional keywords:

`server`	`invoke`	`accept`	`timeout`	
`priority`	`replicas`	`tolerance`	`hosts`	`disjoint`
`model`	`correct`	`crash`	`byzantine`	

and these additional predefined classes:

`host`	`future`

Chapter 4

Design

In this chapter we aim to show that the techniques of the state machine approach presented in chapter 2 can be implemented and used by programs written in the Concurrent MSL (CMSL) language presented in chapter 3. Our approach has two parts:

1. A high-level transformation part in which CMSL programs are modified and finally compiled into C++ .
2. A low-level runtime support part which encodes the techniques of the state machine approach.

The runtime support includes protocols for coordinating replicated servers over a network of distributed processors. The CMSL program transformation uses this runtime support to ensure fault tolerance.

Section 4.1 discusses the high-level transformations. Section 4.2 presents some design options for the runtime support and indicates which options we chose to implement. We conclude with a description of the prototype system that we built. Section 4.3 tells which features of CMSL from chapter 3 and which transformation and runtime support features from chapter 4 were actually implemented during the project.

4.1 Transformations

Three kinds of CMSL program transformation appear to be most useful.

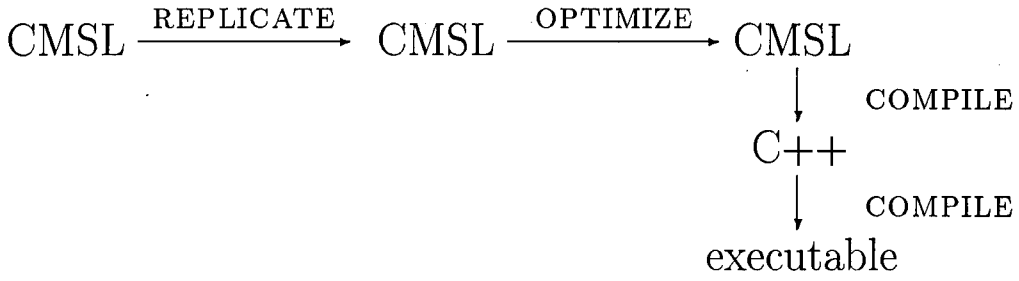


Figure 4.1: CMSL transformations

1. *replication:* This transformation turns an MSL object or CMSL unreplicated server into a fault tolerant server. The transformation is as simple as converting an MSL expression involving `new` into a CMSL expression involving `server` with specifications for fault tolerance. The textual transformation is trivial, but the difficulty comes in implementing the fault tolerance specifications. We discuss this implementation in the next item.

2. *compilation:* A CMSL program can be transformed, or "compiled", into a C++program. The compilation replaces the special CMSL language constructs with C++ calls to a runtime support system. The runtime support system in turn calls the local operating system for interprocess communication services.

   One might quibble that this compilation is not really a "transformation" of the code but rather a "translation" of one language into another. Whatever it is called, this step is crucial for running fault tolerant CMSL programs.

3. *optimization:* A CMSL program with remote procedure calls to different servers can sometimes be optimized by breaking the calls into invoke-reply pairs and reordering the pairs. This reordering can increase the amount of parallelism in the program and thus can make the program more efficient.

Of course, these three kinds do not include the wide variety of transformations possible for pure MSL programs. We are considering only new transformations that are useful when MSL is extended into CMSL.

The three transformation kinds are shown schematically in figure 4.1. We assume that the programmer is able to choose the fault tolerance specifications appropriately, so the replication transformation needs no special support. In the next section we discuss compilation first, then optimization.

4.1.1 Compiling CMSL

Executables

Compiling a CMSL program results in a set of executables, not just one as for an MSL program. Different executables are needed for any of several reasons.

In the simplest case two different executables are necessary: one for the processor that will begin executing the "main" program and one for "subordinates", i.e., all other processors that will eventually execute server replicas started by the program. All server replicas are started by the main program, either directly or indirectly by other servers. Thus the subordinate executables start out simply idling, waiting for the first server replica to be started.

More complex cases can require more than two executables. If the network that is to run a distributed CMSL application has host processors with differing instruction set architectures, or with differing operating systems, then obviously different executables must be produced for each kind of host.

Even if all processors have the same architecture and operating system, the design of the CMSL runtime system may require that several different executables run on each host. For example, if each server replica is run as a separate operating system process then a separate executable is needed for each server replica. In this case the runtime support must also compile into its own executable for each host. Thus the number of executables produced by the CMSL compiler depends on the design of the runtime system. We discuss the design of the CMSL runtime system in section 4.2.

Startup

How are the different executables started when needed? The answer to this question depends on the runtime environment. If the host operating system provides a means for distributing executables over the network (e.g., the Sun Network File System) then one of the first responsibilities of the main executable is to bring up the CMSL runtime system on all other hosts needed in the application. If no such means is conveniently available (note that a protocol for transferring executables could always be built using the network facilities) one can start the CMSL runtime system by hand on each network host.

Program Analysis

What information about a program must the CMSL compiler collect? This question is important because it influences the choice of tools used for program transformation. If a lot of information must be collected about a program, then the transformation tools must be able to do a sophisticated analysis of both the program's syntax and semantics. If only a little information need be collected, for example, if expressions that use CMSL keywords can simply be rewritten into calls on the runtime system with little knowledge of the context in which the expressions are used, then the compiler can be a simpler transformation tool such as a preprocessor. We consider each of the CMSL constructs, sketch how it will be compiled, then discuss how much context information a CMSL compiler must collect to transform it correctly into MSL.

To convert a `server` expression into a call on the runtime system the compiler must analyze all the `server` expressions in the program to create a table of server class names. Each `server` expression then is transformed into a call with the appropriate name. Each call is also parameterized by the fault tolerance properties specified in the `server` expression. The compiler must introduce a new class for server instances, i.e., local objects that are created by `server` expressions to represent each distributed server, and one of these local objects is returned by the runtime system call. Finally the compiler builds the table of server class names into the runtime support system so that a request to start a server replica can be carried out remotely. This transformation can be used to transform all `server` expressions in a program with no knowledge of context in which the expressions appear.

Unfortunately, the compiler still needs information about the context of each `server` expression for other reasons. CMSL puts constraints on which MSL classes can be named in `server` expressions (section 3.3.1). A server may not share variables with its client or other servers. To enforce this constraint a CMSL compiler must inspect the program for declarations of variables used in every `server` class. This inspection gathers a great deal of context information and therefore cannot be done with a simple preprocessor.

To convert an `invoke` expression into a call on the runtime system the compiler must

- supply the server instance and method names as parameters;
- supply the method arguments as parameters;
- return a `future`.

A difficult question for the implementation is: how can the argument data be packed

for transmission to the server replica and unpacked after reaching the replica? This question is not new; the problem must be solved for every language with remote procedure calls. Obviously this packing and unpacking can only be done using type coercion because an `invoke` may be given arguments of any type whereas the type of arguments to the runtime support call must be declared in advance. The simplest solution is for the packing operation to coerce each argument to an array of bytes. The packing must also be given the names of classes being packed so that the unpacking, the inverse operation, can coerce the byte array back into objects of the appropriate classes.

In general, the packing and unpacking operations for invocation arguments will require that the transformation tools analyze the program to find the declaration for each method invoked and extract type information from it. If a type refers to other types, those type declarations will need to be analyzed as well. Therefore the conversion of `invoke` expressions will need a lot of context information and cannot be carried out by a simple preprocessor.

Converting an `accept` expression into a call on the runtime system is straightforward and requires no context information. The optional `timeout` clause is simply transformed into an optional parameter to the call.

Note that the overall strategy of compiling CMSL into C++ by transformation leaves the problem of type checking entirely to the C++ compiler. This fact simplifies the CMSL compiler's design.

4.1.2 Optimizing CMSL

The technique of replacing synchronous remote procedure calls with asynchronous `invokes` and `accepts` was the motivation behind the language constructs in section 3.3.3. This kind of transformation has the potential to speed up a program by doing several tasks in parallel rather than in sequence. In general this transformation consists of the following steps:

1. Identify the remote procedure calls to be made asynchronous. These calls must be made on servers of the same class. The server instances, though, should be different and should be running on different hosts, and no command should depend on or affect the results of any other because otherwise the calls cannot be made parallel. This step is done by the programmer because no compiler can decide, in general, if two variables that name servers are distinct and their commands independent.

2. Declare an array of `futuress`, one for each server.

3. Split each remote procedure call into an `invoke` and `accept`.

The example in section 3.3.3 showed the result of this kind of transformation. Here we show the code both before and after the transformation. Consider the following declaration for a class of tracks in an air traffic control database, where each track consists of a set of points at which an aircraft was detected.

```
class track {
  public:
    boolean add_point (point p);
}
track tr[5] = server track [5] disjoint;
boolean added[5];
point p;
int i;
```

This code for adding a new point to every appropriate track

for every `i`, $0 <= i < 5$,

```
    added[i] = tr[i]->add_point (p);
```

is transformed to this code

```
action act[5];
```
for every `i`, $0 <= i < 5$,

```
    fut[i] = invoke tr[i]->add_point (p);
```

for every `i`, $0 <= i < 5$,

```
    index = accept (fut, 5);
    added[i] = fut[i]->result;
```

4.2 Runtime Support

A CMSL program needs a substantial amount of infrastructure in place before it can be run. This infrastructure, the runtime support, includes

- interhost communication
- multiprocessing on a single host
- interprocess communication within each host
- synchronized clocks
- algorithms to implement the state machine approach: order, agreement, voting, and reconfiguration.

This section proposes a layered structure for the runtime support. We do not claim that the structure we propose is the only workable structure, but we will make some comparisons between our structure and its alternatives and will justify the design choices we propose.

Perhaps the biggest influence on the structure of a runtime support system is the off-the-shelf software and hardware assumed to exist already. If a lot of support exists already, then less support software needs to be written from scratch. On the other hand, if a lot of off-the-shelf support is used, the possible designs for the runtime system are fewer. The easiest components to take directly off-the-shelf are the host operating system and the interhost communication facilities, and so we discuss these next.

4.2.1 Host Processes

Each host must run one or more server replicas plus the overhead processing needed to support the state machine approach. Some of this overhead, such as maintaining a synchronized real time clock, will be shared by all the server replicas and so will most easily be handled separately from any of them. We will call the shared overhead processing the *host administration*. More will be said about host administration later.

Most host operating systems support multiple *processes*, so one easy design decision is to run all server replicas and the host administration as separate host processes. The alternative is to run all replicas and the host administration in a single host process,

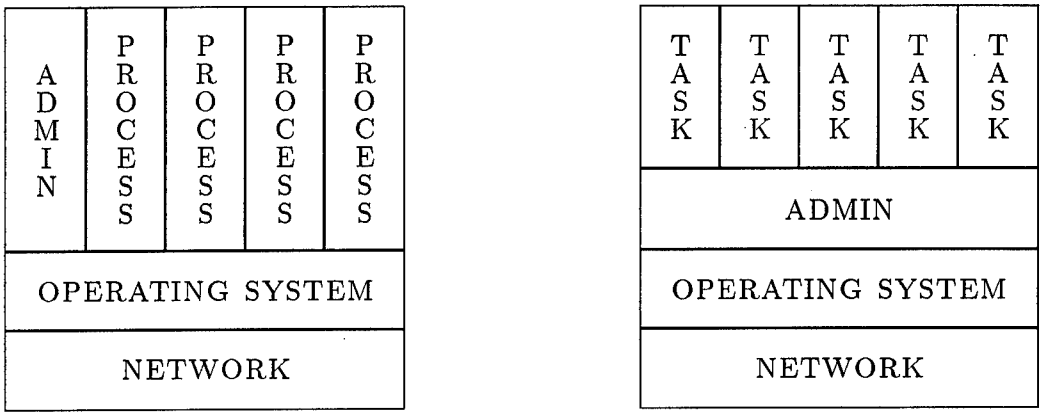


Figure 4.2: multiple processes or multiple tasks?

but within that process create separate subprocesses, i.e., *tasks*, for each replica. These alternatives are depicted in figure 4.2. Deciding between these alternatives must take into account a tradeoff, and we list the disadvantage of each:

- *Alternative #1:* Using multiple host processes would require each server replica and the host administration to use the host's facilities for interprocess communication. These communication facilities are likely to be slower than intertask communication implemented in alternative #2, which can more easily take advantage of shared memory communication.
- *Alternative #2:* Using a single host process to create a CMSL environment on the host would require substantial modifications to the C++ compiler while alternative #1 requires no such modification. The compiler must be modified to maintain a separate state for each task. Furthermore, the state of the various tasks will probably be maintained without using any hardware support for multitasking (the operating system probably uses such support, if it exists, for multiprocessing). Without hardware support to keep tasks isolated from each other, incorrect code in one task could clobber code or data in other tasks.

We view the drawbacks of alternative #2 to be much worse than those of alternative #1. Thus we plan to use separate host processes to run each server replica. Further supporting this decision, many host operating systems provide some means for shared memory interprocess communication so it is possible the efficiency problem mentioned for alternative #1 can be overcome on these hosts.

One special case deserves mention. If a host is running exactly one server replica then that replica can be combined with the host administration into a single host

process. In this special case there is no need for the replica's command processing to be separated from the host administration, and so there is no need for tasks or intertask communication: the host administration can simply call the replica's command processing as subroutines. Therefore in this special case we can combine the advantages of both alternative #1 and alternative #2.

4.2.2 Host Administration

What exactly is the host administration? At the very least it has these responsibilities:

- It handles requests for creating and destroying server replicas. To create a replica it must ensure that the replica is assigned a unique identifier, has the proper event handlers running, is initialized with any startup data such as the current logical clock time, and has the proper connections for interprocess communication.
- For some protocols, it must maintain the physical real time clock by synchronizing it with the real time clocks maintained on other hosts.

If that is all the host administration does, then each server replica must be responsible for directly communicating with its clients and with its peers on other hosts, and for establishing agreement with its peers on the commands invoked and on the order of those commands. In other words, if the host administration is minimal, then most of the work needed to implement the state machine approach must be built into the state machine replicas.

The alternative to minimal host administration is to put all of the overhead of the state machine approach into the host administration. Then the host administration must establish agreement and order for client commands for every server replica on its host, and every replica is dependent on the host administration to provide it this interface to the rest of the system.

As usual, a tradeoff presents itself and we list the advantage of each alternative.

- *Minimal host administration:* Communication between client and server replicas can go on directly without involving the host administration. These processes simply call the network communication facilities. In contrast, maximal host administration requires the host administration to act as an intermediary between communicating processes, as shown in figure 4.3. Thus minimal host administration has an advantage because fewer communication steps are needed between client and server.

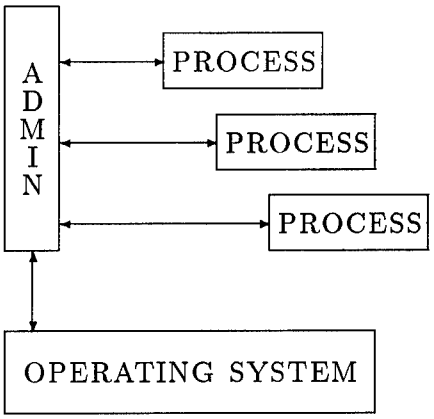


Figure 4.3: all communication via the host administration?

- *Maximal host administration:* In this case all the machinery of the state machine approach is isolated in a single host process. The priority of this host process can then be raised above that of the server replicas so that ordinary command processing does not compete with the machinery of agreement and ordering protocols for the use of the processor. This design is used with success in [14], where the host administration is not only given priority but is even given specialized support in hardware.

  Isolating the machinery of the state machine approach is essential in protocols that depend on the existence of a real time bound on communication delays. These protocols assume a limit on the interval between a client's sending a command and the server's receipt of it. To implement such a real time bound, each host must guarantee its ability to forward messages promptly, and this guarantee is much more difficult if the message-forwarding machinery is scattered throughout every server replica and must compete with the other processing going on in those replicas.

Designs that are intermediate between minimal and maximal host administration might exist, but we have not investigated this possibility.

We have chosen the maximal host administration design because the real time requirements that support fault tolerance must be met, whereas the performance enhancements in the minimal host administration design are optional. Also, some performance enhancements might be achieved even in the maximal approach if shared memory interprocess communication is used.

4.2.3 Networking

The design of a modern computer network is almost always layered. As explained in [30], network layering is now standardized in a model with 7 protocol layers by the International Standards Organization (ISO). While few networks in common use follow the ISO protocols exactly, most protocols have functions similar to those found in the ISO layers.

The 7 ISO communication protocol layers are as follows, from lowest (i.e., closest to the physical hardware) to the highest:

1. *the physical layer* determines how data bits are moved from one host to another by directly using the physical communication medium, e.g., wires, coaxial cable, or optical fiber.

2. *the data-link layer* breaks the physical layer bit stream into chunks and checks for loss or corruption of the chunks.

3. *the network layer* moves the chunks of the data-link layer from a sender host to a receiver host, and if the sender and receiver are not connected by a single data-link then a route using several data-links is found.

4. *the transport layer* moves data from a sender process to a receiver process not necessarily on the same host, ensures that the data is always delivered, and delivered in the sequence it was sent.

5 *the session layer* adds functions, such as checkpointing and recovery, to the transport layer facilities.

5. *the presentation layer* converts data from its representation on the source host to its representation on the destination host.

6. *the application layer* consists of application-dependent communication protocols.

Two very common examples of network protocols are the Transport Communication Protocol (TCP) for layer 4 and the Internet Protocol (IP) for layer 3. Combined as TCP/IP these protocols form the basis for communication in the Internet.

The questions we must answer in designing a runtime support system for CMSL are: Should the ISO layers form part of the design? If yes, which layers?

A hard-line answer to the first question is simply: No, the ISO layers are a good conceptual model for discussing some problems in the design of network protocols but they do not address all the problems that must be solved in a fault tolerant distributed system [16]. In particular, the ISO protocols are not intended to ensure real time bounds on communication delays. Because practical fault tolerance protocols *always* depend on such real time bounds, either for clock synchronization, for determining the stability of client commands (see chapter 2), or simply for detecting that server replicas have failed, the ISO protocols are not adequate for fault tolerant systems.

A more optimistic answer is: ISO-like protocols can be used to meet real time bounds on communication delays if

- the network hardware provides enough redundancy to tolerate a limited number of failures, and
- the network traffic density is limited, and
- each host involved in forwarding messages can complete the forwarding in a bounded amount of time, and
- the network delivery time is known in the absence of failures.

This answer requires the designer to solve some difficult problems of quantifying traffic density, ensuring real time response of each host, measuring network parameters, and analyzing the entire system to prove that the network design has the desired real time properties [9].

A variant on the second answer is: a network using ISO-like protocols need not *guarantee* a real time bound on delivering messages if the probability of delivering a message late is small enough. A late message could cause a server replica to process commands in the wrong order, which could cause the server to fail, which could cause the entire application containing the server to fail. But this probability of server failure must be quantified and is added to the probability of the server failing by any other combination of circumstances. If the probability is small enough, the design is still acceptable [18].

This question of real time support for fault tolerant distributed systems is an open research topic in the fault tolerance community. We believe that addressing it fully is outside the scope of our project. Therefore, we are adopting the third answer above: existing network protocols can be good enough for use in the CMSL runtime support if the resulting probability of system failure is small enough. This approach is not ideal but is the most practical given the time we have available.

Which ISO layers do we need to support a distributed fault tolerant application? One answer is all of them, i.e., a distributed fault tolerant application is a protocol at ISO layer 7 and it uses protocols at every lower layer. This answer is conceptually simple because it decouples the problems of process-to-process communication from the more abstract problems of client-to-server communication. The latter is an example of distributed programming that can be completely separated from the programming of communication between pairs of processes.

A strong objection to using all the ISO layers is that some of the problems solved in programming the state machine approach can be solved more efficiently if the solution were incorporated into relatively low-level ISO protocols. The two most important examples are

1. *detecting failed hosts:* some protocols in the state machine approach require knowing which hosts have failed. For example, if a host has failed other hosts should not wait to receive messages from it. Detecting a failed host will involve at least noticing when the host has taken so long to send an expected message that it has probably crashed. (Other detection methods may also be possible, but a "timeout" is the basic one.)

   Detecting a timeout at the highest ISO layer is inefficient, though, because timeouts are already used for the same purpose at lower layers. For example, the network layer (layer 3) may use timeouts to determine how best to route messages, and the transport layer (layer 4) may use timeouts to determine when to request retransmission of a lost message. Either the state machine protocols should be integrated with lower ISO layers or the information about failed hosts that exists at lower layers should be made available at layer 7.

2. *multicast:* a client invoking a command on a replicated server is an example of a multicast, in which the same message is sent to every one of a group of receivers. One way to implement a multicast is to send a *separate* message to each receiver. In nearly every situation, though, that way to implement multicast is inefficient. The worst case is a multicast over a broadcast medium such as a bus, token ring, or ethernet: the message need only be broadcast once since every receiver can hear the broadcast, but using the separate message method the broadcast would be repeated as many times as there are receivers.

   For efficiency, a multicast primitive should be built into the network protocols at a low ISO layer and made available to higher layers. Modifications to TCP/IP with this goal have been proposed. Multicasting in IP was designed in [11]. The multicast backbone for communicating audio and video on the Internet (MBONE) uses an IP routing protocol described in [34].

Like IP itself, neither of these ISO layer 3 multicast facilities offers *reliable* communication, i.e., they guarantee neither that messages arrive, nor that they arrive in order. These properties are the responsibility of ISO layer 4, the transport layer. A transport layer protocol for reliable multicast was designed in [1] but has not seen widespread use.

We have chosen to replace TCP, which is a reliable, sequenced, point-to-point protocol with one or more reliable, sequenced, multicast protocols. These multicast protocols generalize TCP and allow coordination of fault tolerant servers. Because TCP is roughly ISO layer 4, our runtime support begins at layer 4 also. Like TCP, our multicast protocols use the facilities of IP, which is roughly ISO layer 3. We believe that replacing TCP allows the best use to be made of IP, including the option of basing reliable sequenced multicast on the existing multicast IP mentioned previously.

The layering of our design is as follows, from top to bottom:

- **application:** An arbitrary application can be written in CMSL. This layer corresponds to ISO layer 7.
- **client/server:** invocations and replies are handled at this layer. Code for remote invocations is generated automatically from the CMSL source and linked into the application. This layer corresponds to ISO layer 6.
- **multicast:** reliable, sequenced, many-to-many message passing is handled at this layer. One kernel per host implements the multicast protocols. This layer corresponds to ISO layer 4.
- **operating system:** The local operating system provides facilities for multiprocessing, interprocess communication, and interhost communication.

4.2.4 Clocks

In the state machine approach every server replica must process the commands it receives in the same order. The protocols used for ordering the commands all depend on having a distributed clock for timestamping commands. We now describe two different methods for building a clock as part of the CMSL runtime support. We will implement both.

Distributed clocks come in two kinds:

1. *physical:* a physical clock on each host keeps the real time. Clocks on different hosts are kept synchronized to within a known difference, called the *skew.*

2. *logical:* each host keeps a counter, called a "logical clock", that records whether one message might have causally affected another.

Each kind of clock is used with a different kind of ordering protocol.

A distributed logical clock can be maintained using a simple protocol, described in [17], that updates the clock each time a message is sent or received. In contrast, maintaining a distributed physical clock requires a more sophisticated protocol. Physical clocks will be our concern in the rest of this section.

The basic idea in synchronizing physical clocks is that each host will collect information about the state of clocks on other hosts and use that information to adjust its own clock. The information can be collected at variable times, or it can be collected all at once at predefined times when all hosts resynchronize their clocks simultaneously. Once the information about other clocks is collected, synchronization can be as simple as averaging the difference between the local clock and each other clock, and using that average as the clock adjustment.

If some hosts can fail, other hosts may need to adjust for their failure in the synchronization protocol. The trickiest case is the Byzantine failure model, in which hosts and their clocks can fail in arbitrary ways. In this case averaging is not good enough because a failed clock can give out an arbitrary time when read. It can even present wildly different values when read by different hosts, causing the averages taken by those hosts to diverge.

Many algorithms for Byzantine clock synchronization have been proposed [24]. The Interactive Convergence Algorithm (ICA) is an example [19]. ICA handles Byzantine failures by taking an "egocentric" average of other clock values: clock values differing by more than some cutoff are discarded and replaced in the average by the host's own clock reading. [26] gives a proof that this averaging method works for Byzantine faults.

Our design for a runtime support system uses ICA for all failure models. When the failure model for replicas is not Byzantine, the ICA protocol is overly conservative, but using it in all cases does not seem to have a significant performance penalty.

4.3 Implementation

We now describe the features of our prototype implementation, delivered at the end of the project. We note in particular which runtime support features we implemented, and which parts of the CMSL language we implemented and which parts we did not.

We chose the maximal host administration option, described previously. The host administration was implemented as a single Unix process, called the kernel, on each host. This kernel process has the following facilities.

- The kernel can synchronize clocks on different hosts if protocols need this synchronization.
- The kernel allows sending and receiving of messages between processes on the same or different hosts. The messages can be arbitrary large. Messages can be sent by one sender or many and can be received by one receiver or many.
- Several different protocols for message-passing can be in use at once.
- We implemented a single reliable, sequenced, multicast protocol based on IP.
- The kernel can start a server with replicas on multiple hosts. The replicas can be made to use a specified multicast protocol for coordination.

We did not implement any multicast protocol that needed clock synchronization. We did not implement any way to stop a server or to change the set of replicas that it comprises.

We implemented the transformation of CMSL to C++ as two language processors, one, called "`decl`", for the declarative part of CMSL and one, called "`defn`", for the definitional part. `decl` will parse all of the constructs that existed in the declarative part of MSL as of early 1995. `decl` takes CMSL class declarations and generates the following C++ code:

- C++ class declarations corresponding to the CMSL but containing additional data structures to record the properties of remote objects;
- functions for packing and unpacking an instance of each class to and from a byte array using the XDR external data representation standard[27];
- functions for creating remote objects, invoking methods on them, handling the invocations, replying to the invocations, and handling futures.

`decl` does not handle the following parts of CMSL declarations:

- inheritance;
- collection types;
- relation inverses;
- the builtin string type.

The other transformer, `defn` , takes C++ code with our CMSL extensions and does the following:

- It replaces any expression using the `server` keyword with a C++ call to the appropriate function generated by `decl` ;
- It replaces a call on any method on any server class with a C++ call to the appropriate invocation generated by `decl` .

`defn` does not handle the following aspects of definitional CMSL:

- `server` options other than `tolerate`;
- asynchronous invocation, i.e., the explicit use of futures (synchronous invocation, which we implemented, uses futures implicitly);
- overload resolution.

We did not implement any CMSL optimization of the kind described in section 4.1.2.

By writing a collection of simple Unix shell scripts we solved the problem of distributing both the kernel and the various executables in an application to multiple hosts.

Chapter 5

Fault Tolerance for Large Data Servers

At an early stage of the project, we had thought to do a fault–tolerant distributed file system (FTDFS) as our example application of FT in ADM's state machine approach. The FTDFS we envisioned would have a Unix–like interface consisting of calls to open and close files, and to read and write data from and to files. The FTDFS would be t–fault–tolerant in the sense that it would be implemented on a collection of servers in such a way that as long as no more than t of the servers became faulty, all files would be available to the clients.

It turned out that this application posed unique problems for the state machine approach that the state machine approach needs to be extended to address. Rather than attempt to "shoehorn" an FTDFS into the state machine approach as it stands, we decided to try to define how the state machine approach would need to be extended to handle applications like an FTDFS. While we could not incorporate such extensions into our software on the current project, we feel that it is very likely that users of ADM will eventually wish to use it to develop applications like an FTDFS, so the extensions we've defined, or something like them, will eventually need to be incorporated.

The state machine approach achieves fault tolerance by *replicating* servers. In other words, multiple *copies* of a single, logical server are made, and are run on independent processors. The replicas of a server use special protocols to communicate with each other and with their clients so that the states of the various replicas are kept consistent.

Replicating servers is very desirable from the standpoint of system performance when some of the replicas have become faulty. For example, if faulty processors simply

crash, then the system can potentially function with no decrease in performance at all, as long as there is even a single nonfaulty replica left. This is because each server replica contains a complete copy of all of the data managed by the server.

If the amount of data managed by a server is small, and performance when faults occur is critical, the performance advantage of replication can justify the cost of maintaining multiple copies of the data. For example, if a server replica is simply reading the value from a sensor in an embedded system, the amount of storage needed to store several copies of the value rather than just one is insignificant.

Replication can become extremely costly, however, if the servers manage large amounts of data. This is exactly the case we encountered with the FTDFS. Implementing an FTDFS as a replicated server in the state machine approach would require that we maintain a copy of the entire file system for each server replica. For a file system of any significant size, this may be prohibitively expensive, regardless of the performance benefits. This will also be true of distributed databases of any significant size. We expect large distributed databases to become more and more common in the future, with a concomitant increase in the need for fault tolerance. Thus, it will be important for ADM to support the development of fault–tolerant applications that manage large amounts of data.

In addition to the cost of redundant storage, replication makes inefficient use of processing power when no faults are occurring, because multiple processors are all doing the same work, so n processors are essentially doing the work of 1. If the need for unaffected performance when faults occur is great enough, it may be acceptable to pay the price of inefficient use of processing power when faults are not occurring. When faults are rare, however, and unaffected performance not so critical, we would like to use multiple processors more efficiently by doing different operations in parallel.

The problem of having too much data to simply replicate it (hereafter referred to as the *large data problem*) has already been studied extensively in the domain of disk arrays. There is a large literature on the problem of making disk storage fault–tolerant by storing the data redundantly on multiple disks. This literature has developed a range of solutions to the large data problem that do not require that the data simply be replicated. These solutions are collectively referred to as *Redundant Arrays of Independent Disks*, or "RAID". Our extensions to the state machine approach to solve the large data problem are based on implementing a generic, distributed form of RAID that we call *Distributed Redundant Storage* (DRS). DRS maintains large amounts of data in a fault–tolerant fashion without the high storage overhead of replication. In addition, DRS allows servers to carry out different data transfers in parallel, thus making more efficient use of multiple processors.

Before describing DRS, we will give a brief overview of RAID.

5.1 RAID Overview

The simplest form of RAID achieves fault–tolerant disk storage by taking an array of k *data disks* and adding an additional disk, the *parity disk*. The n^{th} bit on the parity disk is the *parity* of the n^{th} bits of the data disks, that is, the n^{th} bit on the parity disk is the sum of the n^{th} bits of the data disks modulo 2. Thus, the n^{th} bit on the parity disk is 0 if there are an even number of data disks whose n^{th} bit is 1, and 1 if there are an odd number of data disks whose n^{th} bit is 1. The data is therefore stored *redundantly*, since the data on the parity disk is derived from the data on the data disks, but it is not *replicated*.

Any single disk d in such a RAID array can be completely destroyed, and the data on d can be completely recovered from the surviving disks. This is obvious if d is the parity disk, since its data is computed from the data on the data disks. It is also true if d is a data disk, however. To see why, let $d_1, \ldots, d_k$ be the data disks, d_{k+1} the parity disk, and $b_{n,i}$ the n^{th} bit of disk d_i. Suppose d_j is destroyed for some $j \leq k$. We know that

$$b_{n,k+1} = b_{n,1} + \ldots + b_{n,k}$$

(where "+" means addition modulo 2). It follows that

$$b_{n,j} = b_{n,k+1} - b_{n,1} - \ldots - b_{n,j-1} - b_{n,j+1} - \ldots - b_{n,k}$$

(where "−" means subtraction modulo 2). All bits on the right hand side of the last equation can be read from surviving disks, so each bit on d_j can be computed from data in the surviving disks.

Of course, for the above scheme to work, when the data on a data disk changes, the data on the parity disk must be updated accordingly. Specifically, if $b_{n,i}$ changes to $b'_{n,i}$, then $b_{n,k+1}$ must change to $b_{n,k+1} + (b'_{n,i} - b_{n,i})$. In other words, the data on the parity disk must change by the difference between the new data and the old data.

To achieve this degree of fault tolerance using replication, we would need two replicas of each data disk, for a total of $2k$ disks. By using a single parity disk, a RAID disk array achieves the same degree of fault tolerance with only $k + 1$ disks.

The simple parity scheme for recovering from a single failed disk can be extended to handle t failed disks for any t desired. These schemes are called *error detection codes*. A code that can recover t failed disks from the surviving disks is called a *t–error–detecting* code. Such error detection codes are similar to the simple parity scheme in that certain disks are designated as parity disks, and each parity disk holds the parity of a certain set of data disks. The difference is that a t–error–detecting code will have multiple parity disks rather than just 1, and the different parity disks will hold the parity of different sets of data disks. [1]

Higher levels of RAID, such as RAID 5 and declustered RAID, do not put all of the parity on a single disk. Instead, the data blocks on the disks in the array are divided into *stripes*, with each stripe containing $k + 1$ blocks, and each disk containing no more than one block from each stripe. One block in each stripe is the *parity block*, and the remaining k blocks are the *data blocks*. As in the simple RAID, the n^{th} bit of the parity block for a stripe is the sum modulo 2 of the n^{th} bits of the data blocks of the stripe. The advantage of this striping is that the parity can be spread evenly over the array rather than being concentrated on a single disk. This means that the work of updating parity when data is written is distributed over all the disks in the array.

As above, a simple striping scheme with one parity block can reconstruct the data in any single block in a given stripe. It therefore follows that the data on any single disk can be reconstructed from the surviving disks, since each disk contains at most one block from any given stripe. If more sophisticated error detection codes are used for the stripes, the array can withstand the loss of a greater number of disks.

5.2 Distributed Redundant Storage (DRS)

Our approach to solving the large data problem in a distributed fault–tolerant environment is to apply the ideas of striping and error detection codes to the data held by hosts on a network rather than disks connected to a single host. Our solution is, in effect, a distributed form of RAID.

We implement a single logical server as a collection of servers. The data managed by each server is represented as a linear array of bits, which are divided into fixed–size blocks.[2] The blocks on the various servers are assigned to stripes, as in the higher

[1] The number of parity disks needed to recover from t failures varies depending on the number of data disks, but it is always less than would be needed with pure replication, and it is often possible to achieve the theoretical minimum, t parity disks to recover t data disks.

[2] The data in many distributed applications will more naturally be described as an abstract data

RAID levels, and some collection of blocks in each stripe are assigned to hold the parity of other blocks in the stripe, according to some error detection code. If a t–error–detecting code is used, then any t servers can crash, and the data in these servers can be reconstructed from the data in the surviving servers.

So far, DRS looks like RAID with "disk" replaced by "server". There are 2 major differences between centralized systems and distributed systems, however, that will require DRS to work somewhat differently than RAID:

1. Operations in distributed systems can be interleaved in ways that would not occur in a centralized system. In a centralized system, reads and writes by user processes to a RAID must go through the operating system. The operating system carries out such accesses serially and atomically. For example, if process p writes to a data location on disk D, requiring an update to the parity on disk P, and process q subsequently writes to the same data location, the two parity updates will happen in the same order at P as the writes happen at D. In a distributed system, we must take special measures to ensure, for example, that we do not do q's write to D last but p's parity update last, which would cause the data to become inconsistent.

2. In a distributed system, entities operating on data can become faulty, particularly in the middle of an operation that is critical for implementing RAID, such as reconstructing lost data or updating parity blocks. Also, data can be lost in transit between processes and storage. In a centralized system, when such failures occur, it usually means that the entire system fails, so there is no possibility of inconsistency due to some parts of the system failing while others go on.

5.3 Multicast Protocols for DRS

In this Section we will discuss multicast protocols to solve the problems raised at the end of Section 5.2. Throughout this discussion, we will make the standard assumptions made by RAMP, namely that there is an underlying point–to–point message

type or object class than as bit strings. In such cases, the clients and servers dealing with data as objects would have to include a layer of software that would translate operations on objects into operations on bit strings. This situation is no different from the usual situation in higher–order languages that operate on data objects but are implemented on machines whose underlying memories are just arrays of bits.

transmission protocol that will ensure reliable point–to–point delivery of messages, and that will reliably report to the sender if the receiver has crashed.

We will describe the protocols in 3 stages. We will first describe how the protocols would work in the simplified case where there is a single reliable client; the protocols for this simple case form the basis for the protocols for the more general cases. We then describe how the basic protocols must be modified in the case where there are multiple reliable clients. Finally, we describe the modifications that are necessary for the case where the clients can suffer crash faults.

We will assume throughout that there multiple servers, and that servers can only suffer crash faults. We have not dealt extensively with the case where clients and servers can suffer arbitrary Byzantine faults. We discuss more general fault models in Section 5.7.

5.3.1 One reliable client

To read a data block D, the client multicasts a request to each server that holds a block in the stripe containing D, asking it to send the block in that stripe that it holds. If other blocks in the stripe begin arriving before D, the client begins the process of reconstructing the contents of D from the other blocks in the stripe. If, at some point, the client has received enough data to reconstruct the contents of D, it does so, and caches the result locally. If, at some point, the contents of D arrives, the client caches it and discards any other blocks from D's stripe it is holding, as well as any other blocks from D's stripe that arrive in the future.

In any event, if the parity blocks in the stripes are computed using a t–error–detecting code, then as long as at most t servers fail, the client will eventually have the contents of D cached locally.

To write a block D, the client must first have the contents of D cached locally. The client computes the parity difference between the old and new contents (that is, it computes a block of data whose n^{th} bit is the difference modulo 2 between the old and new values of the n^{th} bit of D). The client then multicasts a request to the server that holds D and all servers that hold parity blocks for sets of data blocks that include D. This request is a request to add the parity difference computed by the client to the block in D's stripe held by the server. The net result of this multicast is to change the old contents of D in the server to the new contents (if the server holding D has not crashed), and to update the parity so as to maintain the error–detecting code.

The client may flush the contents of D from its cache at any time. However, it must

read the contents of D back into its cache before doing any new updates, because it needs to know the old contents of D in order to compute parity differences for update multicasts.

Note that one special feature of DRS is that data is updated not by sending the new, updated data to the servers, but by computing the *difference* between the new data and the old data in a certain algebraic structure (n–bit integer arithmetic modulo 2) and sending that to the servers. In fact, this is a necessary aspect of RAID–style redundancy: the parity blocks in a stripe must be updated by adding the difference between the old data and the new data in order to maintain consistency.

This feature has several nice consequences for DRS. First, the client need not issue an update multicast each time it updates its cached copy of a data block D. It can make any number of updates to the cached copy, and simply add the parity difference for each update to a "running total" parity difference. Because the operation of addition of parity differences is associative, updating D with the running total is the same as updating it one difference at a time. Thus, at any time, the client can write all updates out to the servers with a single update multicast using the running total parity difference. In fact, the client need not do any update multicasts at all until it is ready to flush D from its cache.

Second, although we've used the word "multicast" above, these multicasts do not need to be fully atomic. That is, the multicasts do not need to be processed in the same order at all servers. Obviously, different read multicasts can be processed in different orders at different servers and no inconsistency will result. More significantly, *different update multicasts be processed in different orders at different servers.* This is because addition of parity differences is commutative as well as associative. In other words, if C is the contents of a block and U and V are parity differences, $C + U + V = C + V + U$. Thus, applying two different updates in different orders at different servers will yield the same result.

The only degree of atomicity that is necessary for DRS is *between* read and update multicasts. It is necessary that if a read multicast R is processed before an update multicast U by some server, then R is processed before U by every server in the stripe. To see why this atomicity is necessary, consider the following scenario. The client reads a block D into its cache. It makes a change to the cached copy of D, and then sends an update multicast U for that change. Before U has finished, it begins a read multicast R for another block E in the same stripe as D. The server that holds E is down, so the client begins reconstructing the contents of E from the other blocks in the stripe. Some of the servers in the stripe apply the parity difference in U to their data before sending it out in response to R, while others respond to R first,

before applying the parity difference in U. As a result, the data the client uses to reconstruct the contents of E is inconsistent: some of the blocks have been updated to reflect the change in D, and others have not. As a result, the reconstructed version of E's contents is incorrect.

Thus, whatever form of multicast is used for reading and updating, it must preserve order between reads and updates, but need not preserve order between different reads and different updates. This is desirable for system performance: if multicasts require less atomicity, then it will more often be the case that servers can process a request without waiting for other servers to "catch up". This decreases server latency, since servers can respond sooner, and decreases server storage requirement, since servers can discard processed requests sooner. We will discuss how to achieve this degree of atomicity in Section 5.4.

5.3.2 Multiple reliable clients

If the DRS has multiple clients, the basic protocol can remain unchanged, but there is one additional problem: multiple clients accessing the same data block. Consider the following scenario: two clients C_1 and C_2 read the contents X of a data block D into their caches. Client C_1 changes its cached copy to X_1, and sends an update multicast back to the servers in D's stripe containing the parity difference $X_1 - X$. Client C_2 changes its cached copy to a different value, X_2, and sends an update multicast containing the parity difference $X_2 - X$. When both update multicasts have been processed by the servers, the contents of D will be

$$X + (X_1 - X) + (X_2 - X) = X_1 + X_2 - X$$

If $X_1 \neq X$, and $X_2 \neq X$, simple algebra will show that this new value is equal to neither X_1 nor X_2. Neither of the clients updates succeeded, and in fact, the value in D is now garbage.

This problem stems from the fact that updates are done as parity differences. One client can update a data block from X to X_1 by sending the parity difference $X_1 - X$, but then the next update from a client must compute the parity difference from the new data, X_1, and not the original data, X. We solve this problem by requiring that a given data block D must be *locked* by a client before that client can send an update multicast for D, and we allow at most one client to have a data block locked at a time. We augment the protocols from the previous Section to implement this as follows.

When a client C wishes to lock a data block D, it multicasts a request to lock D

to the servers in D's stripe. If another client already has D locked, the request is denied (and C is informed that it has been denied). If D is unlocked, the servers in the stripe record the fact that D is now locked by C.

When a client is finished updating a data block, it multicasts a request to unlock the block. If the client previously had the block locked, the servers mark the block as unlocked.

In this scheme, as in most distributed mutual exclusion schemes, the servers do not actually *enforce* mutual exclusion in the sense that they will reject an update multicast from a client unless that client has the data locked. The servers merely provide the primitives for the clients to arrange mutual exclusion between themselves. This helps performance, as we discuss below.

How much atomicity must the new multicasts have? Clearly, the order in which lock and unlock multicasts are processed must be the same at every server. It might seem like update multicasts must also be processed in the same order with respect to locking and unlocking at all servers to ensure that multiple clients are not reading and updating a given block at the same time. It turns out, however, that the only atomicity needed is between locking and unlocking. To see why it is unnecessary to preserve order of processing with update multicasts as well, consider the following scenario. A client C_1 locks a data block D, then reads its contents, modifies the contents, and makes an update multicast U for the modification. Some servers receive U and apply it to their data. The client C_1 unlocks D. Another client, C_2, locks D and begins a read multicast before U has completed. Since reads and updates are already processed in the same order at all hosts, C_2 cannot receive the data in D until after the multicast U has finished, even though C_2 has D locked. When C_2 eventually receives the data in D, it will be consistent, and will reflect the data currently in D in the DRS, so future updates by C_2 will have the effect that C_2 intends.

The multicast U can proceed to completion despite the fact that C_2 has D locked because of the fact we noted above, that the servers will not reject messages that are part of an update multicast even if they are from a client that does not currently have the data locked. This approach enhances the performance of DRS by decreasing the degree of atomicity required in the lock and unlock multicasts.

The above scheme does not actually mean that only one client at a time can be accessing a given block. It merely means that only one client at a time can be accessing the copy of the block that actually resides in the DRS servers. We discuss the question of concurrent access to data by multiple clients in Section 5.5.

5.3.3 Multiple crash–faulty clients

If clients can crash, there are 2 new situations that need to be handled that are not handled by the protocols in the previous Section:

1. A client could crash in the middle of a multicast to the servers in a stripe. This is not a problem if the multicast is a read multicast. If the client crashes in the midst of an update multicast, however, and some servers process it while others do not, the data in the stripe will become inconsistent.

2. A client may crash in the midst of a lock or unlock multicast, or may crash after having completed a lock multicast, but without doing an unlock multicast. In any of these cases, some of the servers in the stripe will have a data block locked by a client that has crashed. This can keep other clients from being able to lock (and thereby update) the block.

Most of the problems posed by the above situations will not arise if the multicasts are "all–or–nothing" (AoN) multicasts. A multicast M is AoN if it has the following 2 properties:

1. There is some time t at which it will be true that either $(1)M$ has been received and processed by all servers that have not crashed, or $(2)M$ will never be received by any server that is still up at time t.

2. Once a client has finished initiating M, it will eventually be received by all servers that have not crashed.

A client may crash before it has finished initiating an AoN multicast, in which case the multicast may be lost, and it will be as if the client never did the multicast. If such a multicast is not lost, it will eventually be received and processed by every server that has not crashed. An AoN multicast may complete some time after it was initiated, and a client may initiate new AoN multicasts before an old one has completed. A client must, however, finish initiating an AoN multicast before it can initiate another. Atomic multicasts are AoN, but there are weaker forms of multicast than atomic multicast that are nonetheless AoN.

Clearly, if the clients use AoN multicasts for updates, the data in a stripe will never become inconsistent because some servers process an update and some do not. Similarly, if the clients use AoN multicasts for locking and unlocking, the servers will

never get into a state where some of them have the data permanently locked while others do not.

This leaves the problem of a client crashing before initiating an unlock multicast on some data block it has locked. We have developed 2 solutions to this problem, both of which are variants on the basic idea of having data blocks locked and unlocked by fault–tolerant process groups instead of single client processes, so that data will be unlocked even if the processes holding the data crash.

Cache Groups The first, and in some ways most straightforward, variant is simply to replace the single–process clients of the DRS by fault–tolerant process groups (called *cache groups*) with the same functionality. A cache group would be a replicated process group in the usual sense of the state machine approach. All the data that was formerly managed by a single client would be replicated, including the cached data and the list of data blocks currently locked. Replicated data would be maintained consistently by means of multicasts, and other clients wishing to access cached data would communicate with the cache groups by multicasts.

Cache groups do not suffer from the large data problem because only the data currently in a cache is replicated. At any one time, this would be a relatively small fraction of the total amount of data in the DRS.

Lock Groups The second variant is like the first, except that the fault–tolerant process groups (now called *lock groups*) only replicate the information of which data blocks are currently locked. The actual cached data is managed by a single process. The only real function of the lock group is to unlock the cached data if the process managing the cache crashes. For this reason, the lock groups must monitor the process managing the cached data to see if it has crashed. This can be done simply by having the process periodically send an AoN multicast to the lock group telling it that the process is still running, as long as the process has any data locked.

One way to implement lock groups would be to have the lock group for a data block D be the group of DRS servers for the stripe containing D. In other words, have each DRS server group monitor any process that has locked any data in its stripe. If it detects that the process has crashed, the server group unlocks the data for the client.

The overhead required for lock groups is smaller than for cache groups, since the replicated information for lock groups is small and relatively infrequently updated.

Lock groups require the additional multicasts for a client to inform the lock group that it is still up, but these multicasts will require relatively little network bandwidth since they do not carry much information, and will cause little additional latency since they need no atomicity. The major reason to use cache groups rather than lock groups is if it is important not to lose updates to data that have not yet been multicast to the DRS servers, and this is independent of the question of unlocking data when a client has crashed. We discuss the circumstances in which cache groups might be more desirable despite their higher overhead in Section 5.5.

5.4 Atomicity in DRS

As noted in several places above, DRS multicasts need less than full atomicity. If we could implement multicasts so as to enforce only the atomicity needed and no more, this would decrease server latency and server storage requirements. In this Section, we discuss how DRS multicasts could be implemented to provide only the needed atomicity.

First, some implementations of atomic multicast can be modified to provide only partial atomicity. For example, one way to implement atomicity is to have a token circulating among the servers in a group. When the token comes to a given server, that server can take any multicasts it has received but not yet processed, put them in any order, and broadcast that order to the other servers in the group. We will refer to this broadcast as an *ordering broadcast*. Once this is finished, the server can begin processing the multicasts in the broadcast in the order it broadcast, and the rest of the servers must process these multicasts next in that order as well. If a server has received a multicast that has not appeared in an ordering broadcast yet, it must hold that multicast and not process it until it has either received an ordering broadcast in which that multicast appears, or until it receives the token itself, at which time it can order the multicast itself. If a server receives an ordering broadcast in which a multicast appears that it has not yet received, it can only process multicasts up to the first multicast in the ordering that it has not received. The server cannot process another multicast until it has received the next one in the sequence.

This scheme can be modified to enforce only partial atomicity. A partially atomic multicast would use the same token mechanism, but a server in the group would be allowed to rearrange the ordering of multicasts in an ordering broadcast in any way that did not change the order of multicasts that must be processed in the same order at all servers. For example, suppose a DRS server s_1 received the token and had the following multicasts received but not yet ordered: a read multicast R, 3

update multicasts U_1, U_2, and U_3, a lock multicast L, and an unlock multicast M. Suppose s_1 put the multicasts in the order U_1, U_2, R, L, U_3, M and broadcast them to the other servers in the stripe. Suppose another server s_2 has received U_2, U_3, R, and L, and none of the other multicasts in the broadcast. The server s_2 would be free to reorder the multicasts to, say, L, U_2, U_1, R, U_3, M. This does not change the ordering of locking and unlocking multicasts, nor the ordering between a read multicast and an update multicast. However, with the original ordering, s_2 would be unable to process any of the multicasts until it had received U_1. With the new ordering, it would be able to process L and U_2 before stopping to wait for U_1. Note that there is no way for s_2 to reorder the multicasts so that it can process U_3 before it has received U_1, because there is a read multicast between them in the order broadcast by s_1.

5.5 Concurrent Access to Data

The use of locking in DRS makes it seem that clients cannot concurrently manipulate data in the DRS, but this is not quite true. Only one client can have a given data block from the DRS *cached*, but other clients can concurrently access that block by accessing the cached version rather than the version in the DRS servers. A client C wishing to access a particular data block would first attempt to get it from the DRS. If the block is currently locked by another client, C would receive the identity of the caching client. The client C would then use a separate multicast interface (which we will call the *client–client interface*) to requests reads from and writes to the cached data. If C requested an access to a block that the caching client had flushed from its cache, C would be informed that the data had been flushed, and could then attempt to acquire it from the DRS directly.

Writes could be carried out in the client–client interface by sending the new data rather than a parity difference, because the caching client is not using RAID–style redundancy. The client–client interface would therefore not need to ensure that only one client is caching the data at a time. One client at a time would be caching the data as far as the DRS is concerned, and that client would be responsible for relaying changes in the data back to the DRS. ¿From the point of view of the client–client interface, however, many clients could have the data cached. There is a substantial and growing literature on the subject of distributed cache consistency protocols. This work could be implemented in the client–client interface without having to take into account the use of RAID–style redundancy, because there is no RAID–style redundancy from the client–client point of view. In fact, this is one of the principal benefits of the DRS approach: it separates the problem of concurrent access to data from the

problem of achieving fault tolerance for large amounts of data. DRS uses RAID–style redundancy to solve the latter problem, and allows the former problem to be solved independently.

There is one way in which the client–client interface has an impact on the DRS. Suppose a client C has a data block D locked, and a different client C' has its own copy of D. Suppose C' makes a change to its local copy, and then sends a client–client multicast to C to make the same change to C's copy of D. Suppose C then crashes before propagating this change back to the DRS. If another client C'' later reads D from the DRS, it will have the value without C''s change. If C' and C'' are part of a cooperating distributed application, and both clients "assume" that they have the same contents for D, the distributed application may perform incorrectly.

How can we avoid this problem? One way is simply to program clients that access the same data in such a way that they do not "assume" that changes made through the client–client interface are propagated back to the DRS. This will generally make it very difficult to program distributed applications, however, because a client could only "assume" that changes through the client–client interface have made it back to the DRS by obtaining a lock on the DRS' copy of the data, reading the data, and seeing that it has been changed. In fact, this won't even work in general, because a client's change may be propagated back to the DRS and then overwritten by a later client's change. This way of avoiding the above problem will not be feasible in many cases.

The DRS and the client–client interface must provide some way for clients to "know" when a change made through the client–client interface is "permanent" in the sense that any client reading the data in the future will see data that has had the change applied to it. We have developed 2 ways to do this.

The simplest solution is to use cache groups rather than lock groups. In this scheme, C would a fault–tolerant process group rather than a single process, and the data in D would be replicated. Thus, as long as some process in C has not crashed, changes made by C' through the client–client interface will eventually be propagated to the DRS.

The other solution is to use lock groups, and to have the process caching the data inform other clients when a change made through the client–client interface has been propagated back to the DRS. In this scheme, C' would not assume that its change was permanent until it had received a message from C saying that C had updated the DRS from its cache since C''s change. This scheme would generally require that clients update the DRS more often when changes are being made through the client–client interface so that other clients could proceed under the assumption that their

changes had been made. Which of these 2 solutions is more desirable depends on the particular applications running. If concurrent access to data is rare (as it generally is in a file system, for example), the scheme using lock groups might be more desirable. If concurrent access is frequent, however, the increased bandwidth due to more update multicasts might outweigh the cost of replicating the cache in a cache group, making the cache group solution more desirable.

5.6 Optimizations

There are several ways in which the DRS multicasts can be optimized that are worth mentioning. For read multicasts, the client can send the request for D itself first, and delay the requests for the other blocks in D's stripe to give the server holding D a chance to respond if it's up. In the common case where the server holding D has not crashed and the network is not too heavily loaded, this will mean that reading incurs no overhead at all for fault tolerance. If the client has begun reconstructing the contents of D, and the contents of D then arrives, the client can actually send out messages cancelling the other read requests rather than waiting for the other blocks to arrive and then discarding them.

If a client wishes to do a write that is specifically to flip a certain collection of bits in a block (if, for example, the contents of D is meant to represent a collection of flags), the client need not read the block first. It can instead simply lock the block, and then send an update multicast consisting of a 1 for each bit the client wishes to flip. This update will be the parity difference between the old and new data in the block, which (in this special case) can be computed without reading the data first.

If the client makes a change to a small segment of a cached block, the parity difference for that block will consist of 0's except in the segment where the change has occurred. In such a case, the client need not send an update multicast consisting of a block of mostly 0's, but can instead send just the parity difference for the modified segment alone, along with the segment's offset within the modified block.

Various "combination" multicasts can be made that decrease the number of messages required to carry out various operations. For example, a "lock–and–read" multicast would attempt to lock a data block D and then either send the data in D's stripe to the requesting client, or send the client a notification that the block is already locked. Similarly, an "update–and–unlock" multicast would apply an update to a data block and its parity blocks, and then unlock the block. This would be used, for example, when the client was flushing the block from its cache and some updates

have occurred.

5.7 A More General Fault Model

We have assumed in this Chapter that faulty servers and clients simply crash. It would require significant changes to the DRS protocols to enable them to withstand arbitrary or Byzantine faults. There is, however, a more serious type of fault than simply crashing that DRS can be modified to withstand without significant changes.

Suppose that we can reasonably assume that the software that carries out the DRS protocols will function correctly as long as the server has not crashed, but that the data in the DRS' data blocks can become corrupt even though the server has not crashed. This type of failure can be withstood by using error–*correcting* codes instead of the error–*detecting* codes used above. Formally, a t–error–correcting code is simply a $2t$–error–detecting code. The important fact for tolerating data corruption faults is that if we store data redundantly in the DRS according to a t–error–correcting code, and fewer than t servers in a stripe have corrupt data in that stripe, then the lost data can be recovered from the uncorrupted data in the stripe. Further, this recovery does not require that we know which servers have the corrupt data: the recovery process both identifies the servers that have corrupt data and reconstructs the uncorrupted data.

Since a t–error–correcting code is simply a $2t$–error–detecting code, the DRS multicast protocols described above can be used without change. The only element of the DRS that must be changed is the reconstruction process applied by a client that is reading a data block. The client must apply the reconstruction process for error–*correcting* codes to determine if the desired block is corrupt, even if the desired block has been received from a functioning server.

As we might expect, error–correcting codes require more redundancy than error–detecting codes. This means that more storage space on servers must be devoted to parity in order to tolerate data corruption faults. This additional overhead buys more than merely tolerance of data corruption, however. Since a t–error–correcting code is a $2t$–error–detecting code, a DRS that can tolerate t data corruption faults can tolerate $2t$ server crashes, and can in fact tolerate combinations of server crashes and data corruption faults.

Chapter 6

Conclusion

In this final chapter we summarize our accomplishments to date in terms of the technical discussion of the previous chapters and we list some ways in which our work could be extended.

6.1 Accomplishments

We have extended the ADM Minimal Specification Language (formerly MSL, now Argo) with constructs for concurrent, distributed, and fault tolerant programming, resulting in a larger language called Concurrent MSL (CMSL). CMSL allows a programmer to start servers that execute in parallel with other servers and with their clients, to invoke commands on servers and receive replies, and to specify fault tolerance properties of servers. As much as possible CMSL respects the syntax and style of the MSL on which it is based. We believe that the CMSL extension can be used in ADM for a wide range of concurrent programming as well as to demonstrate program transformations that ensure fault tolerance.

We implemented a significant part of CMSL. Our implementation transforms CMSL into C++ with calls on a runtime support system. That runtime support is layered. The higher layers implement the state machine approach using synchronous and asynchronous invocation, and reliable atomic multicast. The lower layers use Unix for multiprocessing, IP for network communication, and XDR for external data representation.

We implemented an Air Traffic Control demo application which shows the use of CMSL, the use of our program transformation, and the fault tolerance possible with

our CMSL runtime support.

We developed extensions to the state machine approach which can improve its performance when the servers manage large amounts of data. We called these extensions *Distributed Redundant Storage* (DRS).

6.2 Options for the Future

The obvious way in which our work could be extended would be to implement the complete CMSL language. This implementation will be easier when ADM is done because it will be easier to use and extend the parsing and static semantic analysis tools now being built for ADM than to build from scratch as we did.

Other options for future extension of our work include:

- Implement DRS.
- Add other protocols to the runtime support. In particular, protocols that depend on synchronized physical clocks are an important alternative to the protocol we developed, which depends only on logical clocks. We began the implementation of a protocol based on physical clocks but could not complete it during the project. A protocol that tolerates Byzantine failures would also complement the current protocol, which tolerates crash failures only.
- Give the developer additional support for specifying fault tolerance properties. CMSL allows specification of fault tolerance in terms of number of host failures but a grander system would help the developer to relate these specifications to fault tolerance requirements such as a mean-time-to-failure. Specifications describe the software; requirements would describe the entire system that needs to be fault tolerant.

Bibliography

[1] ARMSTRONG, S., FREIER, A., AND MARZULLO, K. Multicast transport protocol. Internet file ftp.uu.net:/inet/rfc/rfc1301.Z, Feb. 1992.

[2] AVIZIENIS, A., AND KELLY, J. Fault tolerance by design diversity. *IEEE Computer 17*, 8 (Aug. 1984).

[3] BAL, H., STEINER, J., AND TANENBAUM, A. Programming languages for distributed computing systems. *ACM Comput. Surv. 21*, 3 (Sept. 1989).

[4] BARBORAK, M., MALEK, M., AND DAHBURA, A. The consensus problem in fault-tolerant computing. *ACM Comput. Surv. 25*, 2 (1993).

[5] BIRMAN, K., AND VAN RENESSE, R. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[6] CARRIERO, N., AND GELERNTER, D. How to write parallel programs: A guide to the perplexed. *ACM Comput. Surv. 21*, 3 (Sept. 1989).

[7] CATTELL, R., Ed. *The Object Database Standard: ODMG–93.* Morgan Kaufmann Publishers, 1994.

[8] CRISTIAN, F. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Symp. Fault Tolerant Computing* (1985).

[9] CRISTIAN, F. Synchronous atomic broadcast for redundant broadcast channels. *J. Real-Time Systems 2* (1990), 195–212.

[10] DEBELLIS, M., ET AL. KBSA Concept Demo. Tech. Rep. 93-38, USAF Rome Laboratory, 1993.

[11] DEERING, S. Host extensions for IP multicasting. Internet file ftp.uu.net:/inet/rfc/rfc988.Z, July 1986.

[12] ECHTLE, K. Fault-masking with reduced redundant communication. In *Symp. Fault Tolerant Computing* (1986).

[13] GRIES, D. *The Science of Programming.* Springer-Verlag, 1981.

[14] HARPER, R., AND LALA, J. Fault-Tolerant Parallel Processor. *J. Guidance, Control, and Dynamics 14*, 3 (May 1991), 554–563.

[15] HOPKINS, A., T.B. SMITH, III, AND LALA, J. FTMP – a highly reliable fault-tolerant multiprocessor for aircraft. *Proc. IEEE 66*, 10 (Oct. 1978).

[16] KOPETZ, H., AND GRUNSTEIDL, G. TTP – a time-triggered protocol for fault-tolerant real-time systems. In *Symp. Fault Tolerant Computing* (1985).

[17] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978).

[18] LAMPORT, L. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst. 6*, 2 (Apr. 1984).

[19] LAMPORT, L., AND MELLIAR-SMITH, P. Synchronizing clocks in the presence of faults. *J. ACM 32*, 1 (Jan. 1985).

[20] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (1982).

[21] MEYER, B. Applying 'design by contract'. *IEEE Computer* (Oct. 1992).

[22] NIEUWENHUIS, L. Static allocation of process replicas in fault tolerant computing systems. In *Symp. Fault Tolerant Computing* (1990).

[23] SCHNEIDER, F., GRIES, D., AND SCHLICHTING, R. Fault-tolerant broadcasts. *Science of Computer Programming 4* (1984), 1–15.

[24] SCHNEIDER, F. B. Understanding protocols for Byzantine clock synchronization. Tech. Rep. 87-859, Dept. Comp. Sci. Cornell U., Aug. 1987.

[25] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990).

[26] SHANKAR, N. Mechanical verification of a generalized protocol for Byzantine fault tolerant clock synchronization. In *Formal Techniques in Real-Time and Fault-Tolerant Systems* (Jan. 1992), Springer-Verlag. Lecture Notes in Computer Science, number 571.

[27] SRINIVASAN, R. XDR: External data representation standard. Internet http://www.cis.ohio-state.edu/htbin/rfc/rfc1832.html, Aug. 1995.

[28] STROM, R., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. 12*, 1 (Jan. 1986), 157–171.

[29] STROUSTRUP, B. *The C++ Programming Language.* Addison-Wesley, 1986.

[30] TANENBAUM, A. *Computer Networks*, 2nd ed. Prentice Hall, 1989.

[31] TANENBAUM, A., AND VAN RENESSE, R. A critique of the remote procedure call paradigm. In *EUTECO Conference* (1988), North Holland, pp. 775–783.

[32] TOY, W. Fault-tolerant design of local ESS processors. *Proc. IEEE 66*, 10 (Oct. 1978).

[33] US DEPARTMENT OF DEFENSE. *Reference Manual for the Ada Progamming Language*, 1983.

[34] WAITZMAN, D., ET AL. Distance vector multicast routing protocol. Internet file ftp.uu.net:/inet/rfc/rfc1075.Z, Nov. 1988.

[35] WENSLEY, J., ET AL. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE 66*, 10 (Oct. 1978).

[36] WILKES, C. T., AND LEBLANC, R. J. Rationale for the design of Aeolus: A systems programming language for an action/object system. In *IEEE CS International Conference on Computer Languages* (1986).

☆U.S. GOVERNMENT PRINTING OFFICE: 1996-710-126-20171

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.